AFRL-IF-RS-TR-1999-71
Final Technical Report
April 1999

TASK-BASED AUTHORIZATIONS

Odyssey Research Associates, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. C929

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-71 has been reviewed and is approved for publication.

APPROVED: Joseph V Giordano

JOSEPH V. GIORDANO
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

TASK-BASED AUTHORIZATIONS

Roshan K. Thomas
Ravi Sandhu
Souvik Das

Contractor: Odyssey Research Associates, Inc.
Contract Number: F30602-95-C-0285
Effective Date of Contract: 28 September 1995
Contract Expiration Date: 30 May 1998
Short Title of Work: Task-Based Authorizations

Period of Work Covered: Sep 95 - May 98

Principal Investigator: Roshan K. Thomas
Phone: (607) 257-1975
AFRL Project Engineer: Joseph V. Giordano
Phone: (315) 330-4199

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Joseph V. Giordano, AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1999	Final Sep 95 - May 98

4. TITLE AND SUBTITLE

TASK-BASED AUTHORIZATIONS

5. FUNDING NUMBERS

C - F30602-95-C-0285
PE - 62301E
PR - C929
TA - 02
WU - 02

6. AUTHOR(S)

Roshan K. Thomas, Ravi Sandhu, Souvik Das

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Odyssey Research Associates, Inc.
33 Thornwood Drive, Suite 500
Ithaca NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFGB
525 Brooks Road
Rome NY 13440-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-71

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Joseph V. Giordano/IFGB/(315) 330-4199

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In this project we developed a new paradigm for access control and security models called task-based authorization controls (TBAC). This new authorization control paradigm is particularly suited for emerging models of computing, especially distributed computing and information processing activities with multiple points of access control and decision making. TBAC articulates security issues at the application and enterprise level. As such, it takes a "task-oriented" or "transaction-oriented" perspective rather than a perspective based upon traditional subject-object distinctions. In TBAC, access mediation involves authorizations at various points during the completion of tasks in accordance with the application logic associated with the overall governing process. In contrast, the subject-object view typically divorces access mediation from the larger context in which a subject performs an operation on an object. By taking a task-oriented view of access control and authorizations, TBAC lays the foundation for research into a new breed of "active" security models. TBAC has broad applicability to access control, ranging from fine-grained activities such as client-server interactions in a distributed system, to coarser units of distributed applications and workflows that cross departmental and organizational boundaries.

14. SUBJECT TERMS

Defensive Information Warfare, Computer Security, Security Policy

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Summary

This research project on task-based authorization controls (TBAC) was undertaken at Odyssey Research Associates from September 1995[1] to April 30, 1998.
Work on the project is divided into the following four tasks:

- Task 1: Model Development (Months 1-5)
- Task 2: Language Development (Months 6-10)
- Task 3: Architecture and Implementation Study (Months 11-12)
- Task 4: Development of proof-of-concept prototype

This final report documents the work undertaken on all of the above four tasks. Specific issues addressed in this report include the modeling and specification of authorizations and authorization policies. Various abstractions and modeling constructs are developed for this purpose. We then discuss the use of visual languages for modeling authorizations. Finally, we present the high-level architecture and other details of our prototype implementation, which we then follow with documentation of software.

[1] In previous reports, we have referred to this project as TBA.

2. Methods, Assumptions, and Procedures

2.1 Introduction

In this introductory section, we discuss the assumptions and motivation for this project, as well as our basic project goals and research directions.

2.1.1 Assumptions and Motivation for task-based authorizations

Organizations are increasingly seeking ways to cut costs and achieve greater efficiency in various business functions and processes. This trend has sparked great interest in business process reengineering as well as automation and computerization. As paper-based information processing systems become computerized, the related authorization procedures will inevitably have to be automated and managed efficiently. The TBAC approach described in this report was motivated by this anticipated need to model and automate authorization and related access controls.

An authorization is an approval act, which manifests itself in the paper world as the act of signing a form. Typically, in the paper world, an authorization enables one or more activities and related permissions. The person granting the authorization usually takes responsibility for the actions that are authorized by the authorization. An authorization, as represented by a signature, also has a lifetime associated with it during which it is considered valid. Once an authorization becomes invalid, organizations require that the associated permissions no longer be available. The implementation of TBAC ideas will lead to systems that provide tighter just-in-time, need-to-do permissions. The TBAC approach also leads to access control models that are self-administering to a great extent, thereby reducing the administrative overhead typically associated with fine-grained subject-object security administration.

From the standpoint of research in security models, the motivation for this project comes from the recognition of the limitations of traditional security and access control models. Given these limitations, our objective has been to develop a new paradigm for access control and security models, which we term task-based authorization controls (TBAC). Initial ideas formulated for TBAC were reported in earlier papers by Thomas and Sandhu [3,4]. TBAC is particularly suited for emerging models of computing, including distributed computing and information processing activities with multiple points of access, control, and decision making. It has broad applicability to access control, ranging from fine-grained activities such as client-server interactions in a distributed system, to coarser units of distributed applications and workflows that cross departmental and organizational boundaries.

TBAC articulates security issues at the application and enterprise level. As such, it takes a "task-oriented" perspective rather than the traditional subject-object one. Access mediation now involves authorizations at various points during the completion of tasks, in accordance with some application logic governing overall processes. In contrast, the

subject-object view typically divorces access mediation from the larger context in which a subject performs an operation on an object.

By taking a task-oriented view of access control and authorizations, TBAC lays the foundation for research into a new breed of "active" security models. We consider traditional subject-object security models as "passive", since they merely store basic access control information, such as *who has which permissions to what objects*, and use this information in a context-independent fashion to answer basic access control requests. An active security model, on the other hand, recognizes the overall task context in which security requests arise and takes an active part in the management of security as it relates to the progress within such tasks.

TBAC will have both broad applicability and significant impact in areas such as the automation of mission-critical command and control scenarios (where authorization sequences need to be carefully controlled), the security management of complex operations in high-assurance client-server environments, and forms-based workflow applications such as logistics management, distributed planning and claims processing.

2.1.2 Overview of task-based authorizations

Figure 1 (a) and (b) shows the fundamental difference between subject-object and task-based views of access control and security. In the subject-object view, the basic entities are subjects, objects, and permissions possessed by subjects to gain access to the various objects. This can be represented in an access control matrix. An access control request thus seeks an answer to a question typically posed as

- Is subject *s* allowed access *a* to object *o*?

Contrast this with a task-based view where we seek an answer to a question such as

- Can authorization *a* be granted to a subject when participating in task *t*?

Notice the shift in emphasis from the permissions of individual subjects to objects to the authorizations for subjects within tasks. This change represents a paradigm shift in the way we think of security and access control.

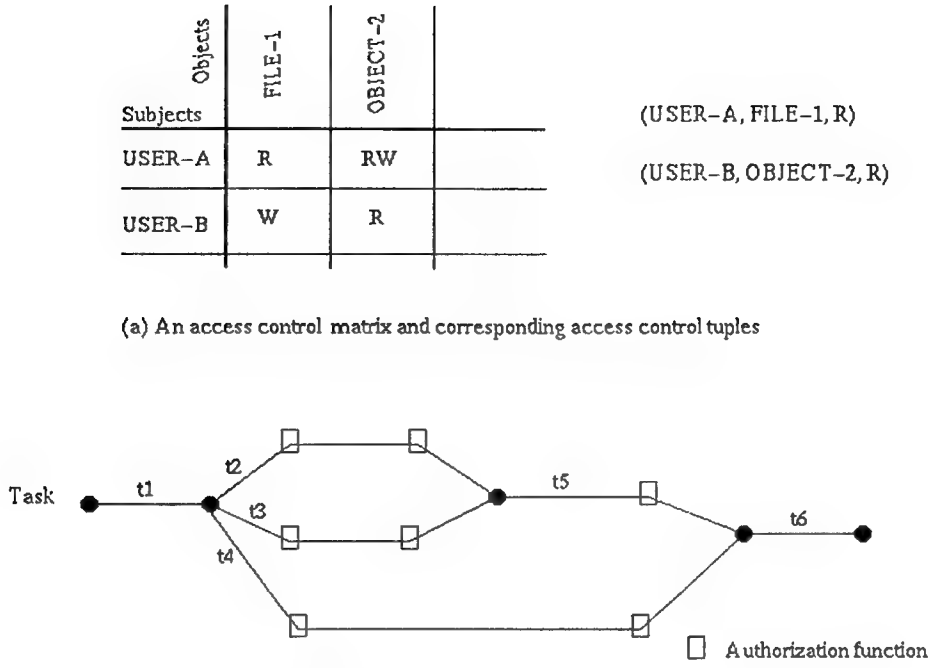


Figure 1. Subject-object versus task-based access control

In a task-based approach to security, there are three basic entities:

- Tasks and sub-tasks: these represent strands of activity.
- Authorizations: these are approval steps that occur at one more points in the lifetime of various tasks and sub-tasks.
- Dependencies: these are relations between authorizations and their encompassing tasks and represent authorization policies.

We are thus interested in how authorizations are modeled and managed as they occur during the lifetimes of various tasks in an information system. These tasks may span the entire workflow/task spectrum. As such, these tasks may be completely automated repetitive processes, or at the other end of the spectrum, represent complex collaborative activities involving humans and automated agents. In the context of TBAC, when we refer to authorizations, we mean the analog of signatures in the paper (forms) world. We see the act of signing a form as an act of authorization (authorization-step). Of course, authorization-steps may be completely automated such as through an automated agent, so long as a human is held responsible for the authorization.

2.1.3 Project goals and scope

We now discuss the goals and scope of this project.

2.1.3.1 Goals and key research directions

In approaching the modeling, specification, and management of authorizations, this project addresses, in a broad sense, four independent but related security objectives; confidentiality, integrity, availability, and accountability. An authorization should be granted only if doing so would not leak confidential information and the associated authorized activity does not adversely affect the integrity of the application or enterprise. Authorizations should be granted in accordance with deadlines and expirations in order to assure availability. Finally, tasks and authorizations should be aligned to the responsibility and accountability structures in the application or enterprise, in order to satisfy accountability.

The key research directions that we have investigated during the course of this project include the following:

- TBAC as an active security model;
- modeling and specification of authorization policies;
- use of visual languages to specify authorization requirements and policies; and
- application of TBAC to distributed enterprise computing and workflows.

During the course of the project, we have investigated several issues concerning the modeling and management of authorizations associated with tasks and the day-to-day activities in an enterprise, including

- interaction of application logic, access control, and task authorizations;
- time and space distribution of tasks involving multiple users, computing resources, and administrative and trust domains;
- authorization for groups of related tasks;
- authorization exceptions and failures;
- dependencies between authorizations and tasks; and
- architectures to implement TBA.

2.1.3.2 Non-goals

The following two area are non-goal areas for the TBAC project.

Authentication: We do not address the issue of authentication, as it is outside the scope of the TBAC model. Users granting authorizations, as well as those receiving authorizations, are assumed to be authenticated elsewhere in the system. TBAC should be seen as a technology for authorization management that is part of an overall enterprise security solution that includes authentication, certificate-based trust management, and other relevant security infrastructure technologies.

Auditing: We also do not address issues related to auditing and audit analysis of authorizations, as our focus is on the modeling and specification of authorizations.

However, our approach to TBAC recognizes the accountability and responsibility structures in an enterprise. As such, our approach can be enhanced to provide appropriate hooks for the future incorporation of enterprise-oriented auditing schemes and tools.

2.2 Security Requirements and Models

In this section, we discuss how TBAC relates to the overall scheme of security requirements and security models.

2.2.1 Abstractions in security requirements

Several levels of abstraction exist when approaching and articulating security requirements and specifications. For example, LaPadula and Williams [2] identify the following stages of requirements:

1. *Trust objectives.* The basic objectives to be achieved by a system.
2. *External interface requirement.* The system's interface to the environment as expressed in terms of the security requirements.
3. *Internal requirements.* Requirements that must hold within the system's internal components.
4. *Rules of operation.* Rules that explain how internal requirements are enforced.
5. *Functional design.* A functional description of the behavior of system components.

The security requirements of a system at stages 1 and 2 above are at a much higher level of abstraction than those at stages 3, 4, and 5. In fact, the higher stages specify *what* needs to be done and these requirements get refined into detailed executable specifications that deal with *how* things are to be done. The higher stages thus involve people-oriented policies and requirements while the lower ones are more computer-oriented.

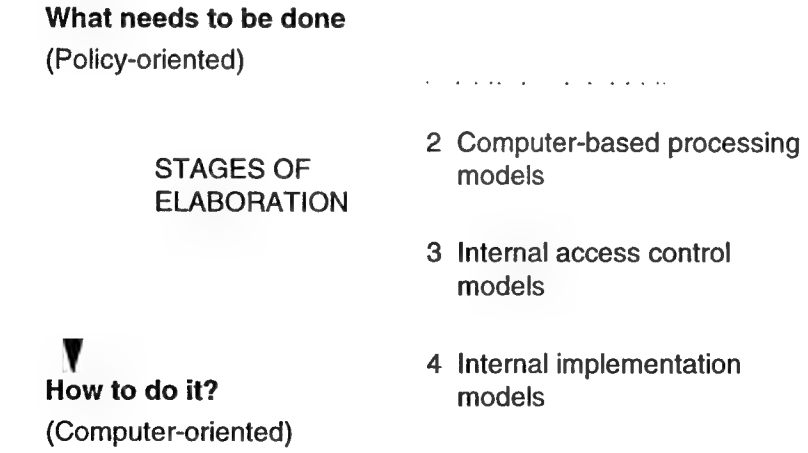


Figure 2. A hierarchy of security models

Given the above stages of elaboration, one can derive a hierarchy of security models such as that shown in Figure 2. As expected, models at the highest level capture organizational policy and requirements that pertain to security and in fact are unaware of the existence of computers. However, at the next stage, these requirements are applied to the interface between the organization and the computer system, and are captured by computer-based processing and policy models. These models recognize and incorporate the abstractions necessary for computer-based processing. Computer policy models, in turn, are implemented by lower level internal access control models that are specific to the abstractions in individual computers; these models, in turn, map to more concrete implementation models, and so on.

Having presented a hierarchy of security models, we next discuss how TBAC relates to such a hierarchy.

2.2.2 TBA: bridging enterprise and system-oriented security models

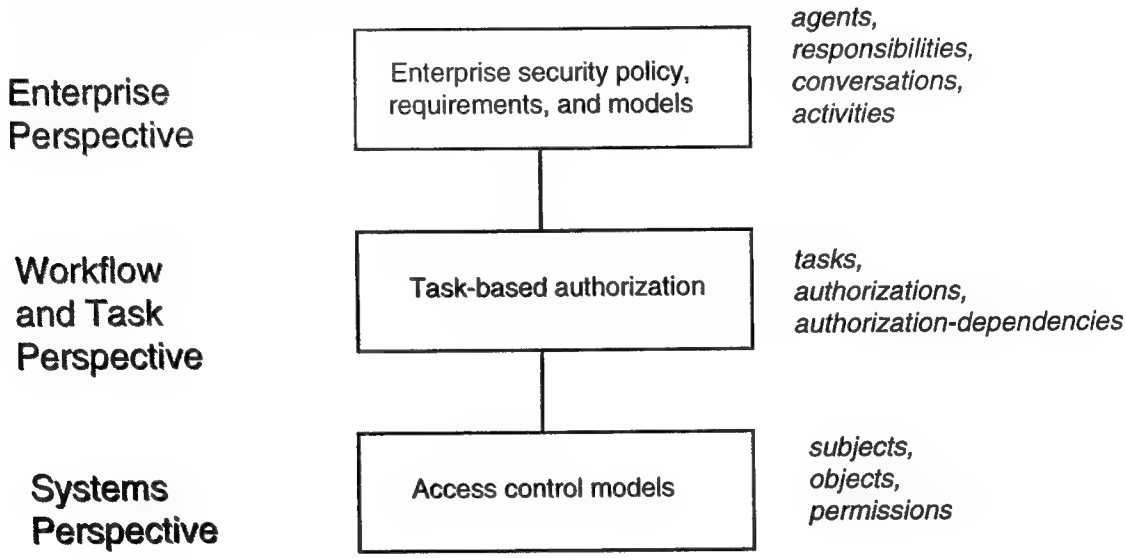


Figure 3. TBAC as the bridge between enterprise security and access control

Figure 3 shows TBAC as the bridge between high-level enterprise security policies and models, and low-level access control models. Traditional access controls and security models, as found in many operating systems and database management systems, have been intensely studied and developed over the last two decades. More recently, a few researchers have started approaching security from the enterprise rather than the system (computer) perspective. Task-based authorizations are unique in this context, as they attempt to link policy and enterprise security requirements to internal requirements and security enforcement.

At this point, it is helpful to understand the motivation for an enterprise-oriented view of security requirements. Traditional approaches to multilevel security, such as those based

on the Bell and LaPadula (BLP), have always taken the view that security is concerned with the internal protection of resources within a computer system. While this is an important requirement, it is becoming clear that these models are primarily machine-oriented and, as such, cannot model, other than in some very limited ways, what security means in the broader context of an enterprise and its mission. After all, a computerized information system is put in place to help achieve organizational objectives. Even if we talk about security in the narrow sense of protection of resources or data in a database, this should be traceable and justifiable with respect to an enterprise security policy, which, in turn, must be mapped to overall organizational objectives.

Recent efforts at enterprise-oriented security modeling have started from enterprise models and investigated how security requirements can be elicited. For example, Dobson et al. [1] identify the following key abstractions as a basis for modeling enterprises:

- Agents
- Responsibilities
- Conversations
- Activities
- Resources

Agents are the people in the enterprise or socio-technical system. Agents hold responsibilities. In fact, in Dobson's view, an enterprise can be seen as a network of responsibilities and a clear understanding of responsibilities is the key to capturing organizational requirements. An agent who is a responsibility holder will inherit a set of obligations that must be discharged in order to fulfill the responsibility. Obligations, in turn, may require agents to execute activities. In other words, obligations form the link between the responsibilities that agents hold and the activities they are allowed to execute. As well, resources are what enable an agent to perform various activities.

Enterprises are not static entities; rather, they are constantly evolving. One can again understand some of these dynamic aspects by looking at responsibilities. An agent, in fulfilling a certain responsibility, may transfer a subset or all of the related obligations to another agent. In Dobson's framework, the responsibility itself can never be transferred. However, the transfer of an obligation to a second agent may result in the creation of new responsibilities. The second agent now becomes a responsibility holder while the original agent becomes a responsibility principal for this new responsibility relationship. As an enterprise evolves, its responsibility relationships change. In fact, the perceived need for this change often triggers corporate restructuring.

The model also introduces the notions of conversations and authorizations. To elaborate, the creation of responsibilities and obligations requires authorization actions. In general, authorization actions, agreements, contract negotiations, and so on are mediated by a process. This process consists of mediation acts called conversations.

It is important to note that concepts such as roles and transactions are missing from the above discussion. This is because the above concepts are an attempt to devise a minimal

set of abstractions from which all others can be derived. Thus the notion of a role can be modeled as a collection of responsibilities. A transaction can be seen as a structuring mechanism to decompose and manage activities.

Let us now see how the above enterprise modeling constructs lead to better elicitation and understanding of enterprise security requirements for computer-based information systems. To start with, it should be noted that associated with every responsibility, a responsibility holder needs to do certain things, needs to know certain things, and needs to record certain things for later audit. The things to do are the obligations and these need to be mapped to functional requirements on the information system. This, in turn, will elicit security requirements for these functional requirements. The "need to know" requirements can help identify security access classes. Similarly, the "need-to-record" requirements will lead to security requirements on the data capture, logging, and audit mechanisms of the information system.

Consider next how conversations lead to security requirements. Conversations at the enterprise level map to business processes, workflows, and protocols in information systems. There might be policies and rules regarding which agents can participate in which conversations. This will lead to access control requirements between processes (or subjects) representing agents and protocol steps. Conversations may also manipulate resources leading to access control requirements between workflows and protocols, and between information structures such as databases. Lastly, conversations may be used to negotiate and acquire access rights. All these have ramifications on security requirements and the design of appropriate security mechanisms.

To summarize, TBAC can form a bridge between enterprise-oriented security models and low level computer-oriented ones. This is possible because TBAC recognizes the abstract notions of responsibility, conversations (processes) and authorizations present in enterprise models and business processes, and maps them to a more suitable model that is amenable to automation and computer-based processing. These notions represent appropriate boundary objects between enterprise security models and system security models.

2.3 Authorization in Tasks and Workflows

In this section, we discuss the relationship between authorizations, tasks, and workflows.

2.3.1 Abstracting the authorization layer in activities and workflows

There are, essentially two perspectives or levels of abstraction in modeling tasks or activities in an organization, as shown in

Figure 4. The enterprise perspective sees activities as business processes. These are abstract descriptions of basic business functions. Below this business process layer, there exists a workflow layer. Workflows are more concrete and detailed manifestations of the business processes that are amenable for computerization.

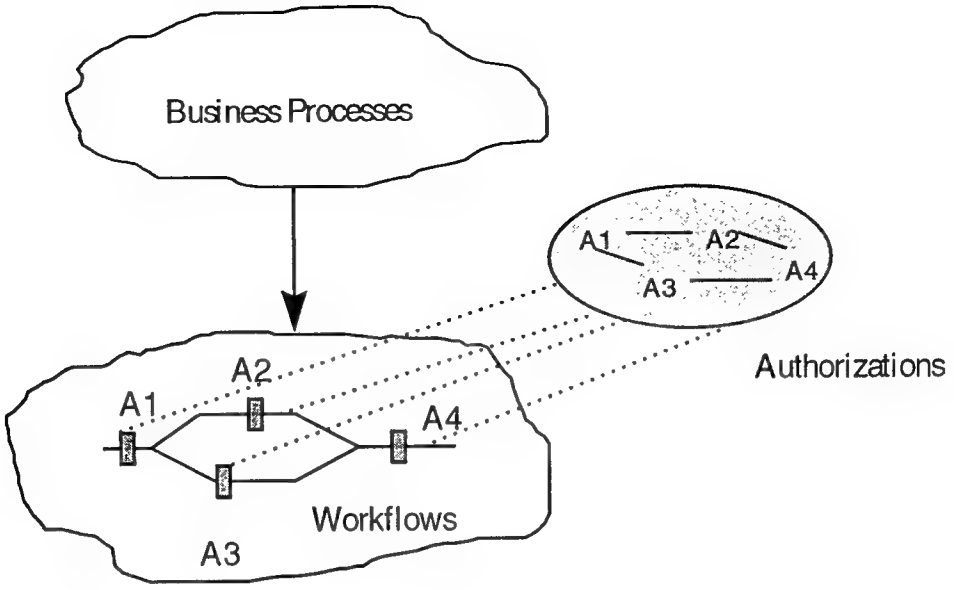


Figure 4. Abstracting authorizations from workflows

Given the above two-tiered structure of activities, task-based authorizations address security at the workflow layer. Our goal is not to reinvent workflow modeling concepts. Rather, given any workflow processing requirement, we want to model and reason about the authorizations that are embedded in the workflow as well as the relationships between various authorizations and tasks. The edges between the authorization steps in the shaded oval in

Figure 4 are meant to convey the fact that authorizations do not stand in isolation. Rather, they are interrelated to each other by various dependencies (explained in Section 3.1.2.3). As mentioned before, we think about authorizations as the analog of signatures in the paper (forms) world. When we mention the term "authorization-step," we are referring to a single (primitive) authorization act, which is the equivalent of a single signature on a form. In Section 3.1.2, we will describe in more detail the attributes (components) that make up an authorization-step.

2.4 TBAC as an active security model for authorization management

We use the concept of active security models to characterize models that recognize the overall context in which security requests arise and take an active part in the management of security as it relates to the progress and emerging context within tasks (activities). Before we elaborate further on TBAC as an active security model, let us discuss some of the basic ideas in the TBAC approach.

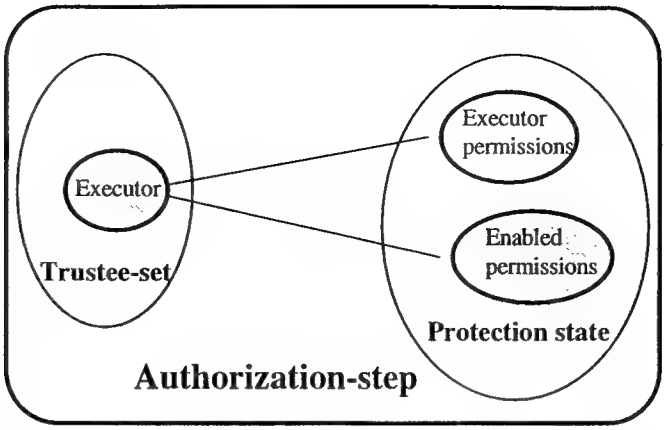


Figure 5. An authorization-step as an abstraction that groups trustees and permissions

One of the most fundamental abstractions in TBAC is that of an *authorization-step*. It represents a primitive authorization processing step and is the analog of a single act of granting a signature in the forms (paper) world. From the standpoint of modeling, it is an abstraction that groups trustees.[2] In the paper world, a group of individuals may be potentially allowed to grant a certain type of signature. For example, all sales clerks may be allowed to sign sales orders. However, a single instance of a signature may be granted by only a single individual. For example, sales order numbered 1208 is signed by sales clerk Tom. Similarly, in TBAC, we associate an authorization-step with a group of trustees[2] called the *trustee-set*. One member of the trustee-set will eventually grant the authorization-step when the authorization-step is invoked and processed. We call this trustee the *executor-trustee* of the step. The permissions required by the executor-trustee to invoke and process the authorization-step make up a set of permissions called *executor-permissions*. In the paper world, a signature also implies that certain permissions are granted (enabled). Similarly, we model the set of permissions that are enabled by every authorization-step. These permissions comprise the *enabled-permissions* set. Collectively, we refer to the union of the executor-permissions and enabled-permissions as the *protection-state* of the authorization-step. Finally, the authority granted by a signature is good for only a limited period of time. Similarly, we associate a period of validity and a life-cycle with every authorization-step.

From the standpoint of access control models, Figure 6 illustrates how the TBAC view of access control differs from classical subject-object access controls. In the latter, a unit of access control or permission information can be seen as an element of the cross product (X) of three domains (sets); the set of subjects, S, the set of objects, O, and the set of actions, A. In TBAC, access control involves information about two additional domains; usage and validity counts, U, and authorization-steps, AS. These additional domains embed task-based contextual information.

[2] We use the term trustee to refer to any one of the following: user, process, agent, service or daemon.

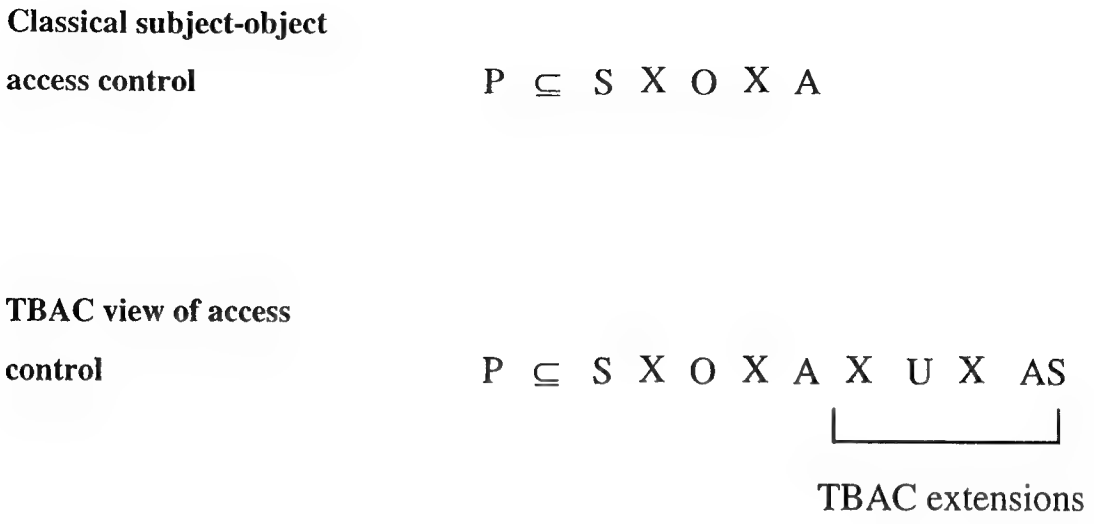


Figure 6. Subject-object versus TBAC views of access control.

In our further discussions, it is useful to be aware of the distinction between an authorization-step class (definition) and an authorization-step instance in a particular workflow instance such as authorize-review in patent workflow instance with identifier 1234 started at 9AM on Dec 1, 1996. We use the term authorization-step loosely to mean authorization-step class or authorization-step instance, as determined by the context. When the context is ambiguous, we will be appropriately precise.

Every authorization-step maintains its own protection state. The initial value of a protection state is the set of permissions that are turned on (active) as a result of the authorization-step becoming valid. However, the contents of this set will keep changing as an authorization-step is processed and the relevant permissions are consumed. With every permission, we associate a certain usage count. When a usage count has reached its limit, the associated permission is deactivated and the corresponding action is no longer allowed. Conceptually, we can think of an active permission as a check-in of the permission to the protection-state, and a deactivation of a permission as a check-out from the protection state.

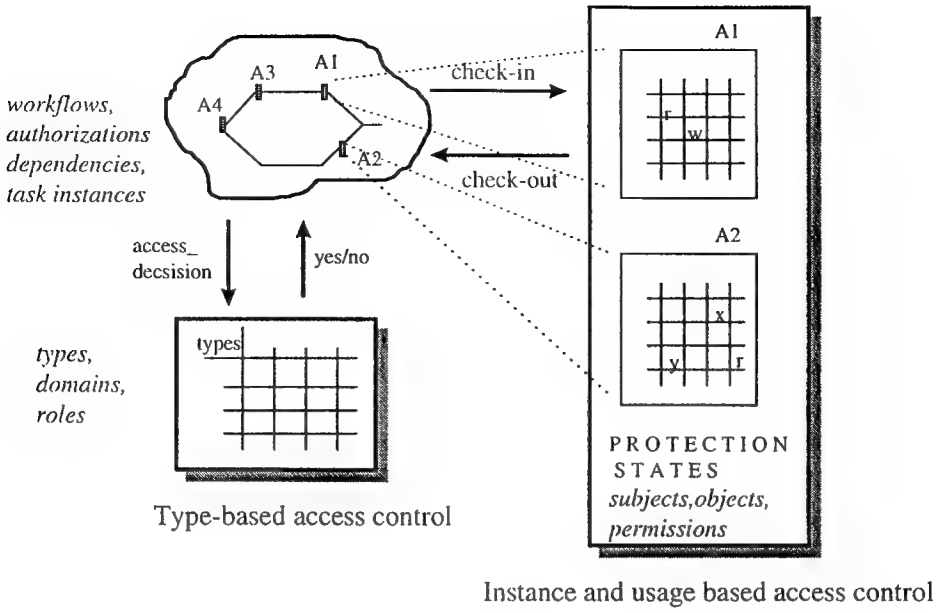


Figure 7. TBAC as an active security model

This constant and automated check-in and check-out of permissions as authorizations are being processed is one of the central features that make TBAC an active model. Further, the protection states of individual authorization-steps are unique and disjoint. What this means is that every permission in a protection state is uniquely mapped to an authorization-step instance and to the task or sub-task instance that is invoking the authorization. This ability to associate contextual information with permissions is absent in typical subject-object style access control models.

The distinction between type-based and instance and usage-based access control is also a significant feature of the TBAC model. Type-based access control is used to encapsulate access control restrictions, as reflected by broad policy and applied to types. Instance and usage-based access control, on the other hand, is used to model and manage the details of access control and protection states (permissions) of individual authorization instances, including keeping track of the usage of permissions.

Figure 7 summarizes the concepts, features, and components that make TBAC an active security model. These include the following:

- the modeling of authorizations in tasks and workflows, as well as the monitoring and management of authorization processing and life-cycles as tasks progress;
- the use of type-based, instance and usage-based access control;
- the maintenance of separate protection states for each authorization-step;
- dynamic runtime check-in and check-out of permissions from protection states as authorization-steps are processed.

To elaborate more on how these concepts are used for the active management of authorizations, consider a simple check-voucher processing example that involves the following sequence of authorization steps (for brevity, we show only the name of the authorization step and the trustee/role that can grant the authorization):

1. authorize_prepare_voucher {clerk}
2. authorize_approve_voucher {supervisor}
3. authorize_issue_check {clerk}

Thus the processing of the voucher involves three phases, namely prepare, approval, and issue. Each phase involves an authorization. As soon as the prepare phase is initiated at the task or workflow layer, there will be an invocation of the first authorization to prepare the voucher (authorization-step 1). This authorization is requested by a clerk, say C. At this point, TBAC will utilize type-based access control and an access decision function to check that entities of type "clerk" are allowed to do the "authorize-prepare" operation on vouchers. If this check succeeds, TBAC will proceed to check-in (or activate) the required permissions so that the specific clerk, C, (who, in this case, is the executor trustee) can do the prepare operation. These permissions are checked into the protection state of step 1. As mentioned earlier, we call these permissions *executor-permissions,* as they permit the executor to process the authorization. As soon as clerk C is done with preparing and authorizing the voucher, we consider the authorize_prepare_voucher authorization to be valid. TBAC will now do two things. First, TBAC will require that previously checked-in executor permissions be checked-out (deactivated) from the protection state of the authorization-step. Next, TBAC may check-in other permissions so as to enable the processing of other activities, including the next authorization-step (step 2), which involves authorization for the approval of the voucher by someone in the role of a supervisor. These permissions make up the *enabled-permissions.* During the processing of this second authorization, the supervisor may consume these checked-in enabled-permissions permissions and, as a result, eventually lead to them being checked-out, and so on. Eventually, when step 1 becomes invalid, all enabled-permissions that are still checked-in (active) will be deactivated (checked-out). Finally, when the third authorization-step authorize_issue_check is invoked, the organizational policy may dictate a separation of duties requirement. In other words, the clerk that is the executor-trustee of the third step will have to be different from the clerk that was the executor-trustee of the first authorization, authorize_prepare_voucher. However, the scope of such a requirement may be limited to only these three authorizations and not to the rest of the authorizations in a workflow. To facilitate such requirements, TBAC supports notions such as *start-conditions* and *scope specifications* that model these kinds of constraints (discussed in Section 3.1.4 below).

In summary, TBAC differs from traditional passive subject-object models by associating the dimension of tasks with access control. First, there is a notion of protection states, representing active permissions that are maintained for each authorization step. The protection state of each authorization step is unique and disjoint from the protection states of other steps. Each authorization-step corresponds to some activity or task within the broader context of a workflow. Traditional subject-object models have no notion of

access control for processes or tasks. Second, TBAC recognizes the notion of a life-cycle and associated processing steps for authorizations. Third, TBAC dynamically manages permissions as authorizations progress to completion. This again differs from subject-object models where the primitive units of access control information contain no context or application logic. In addition, TBAC understands the notion of "usage" associated with permissions. Thus, an active permission resulting from an authorization does not imply a license for an unlimited number of accesses with that permission. Rather, authorizations have strict usage, validity, and expiration characteristics that may be tracked at runtime. In a typical subject-object access control model, a permission associated with a subject-object pair implies nothing more than the fact that the subject has the permission for the object. There is no recognition or monitoring of the usage of that permission. Finally, TBAC can form the basis of self-administering security models as security administration can be coupled and automated with task activation and termination events.

3. Results and discussion

3.1 The Modeling and Specification of Authorizations

3.1.1 A Family of TBAC Models

Rather than formulating one simple monolithic model of TBAC, we have chosen to formulate a family of models. Before discussing the models, we first lay out a framework to guide us in designing the family of models.

Our framework consists of formulating a simple model of TBAC called $TBAC_0$ and using this as a basis to build other models.

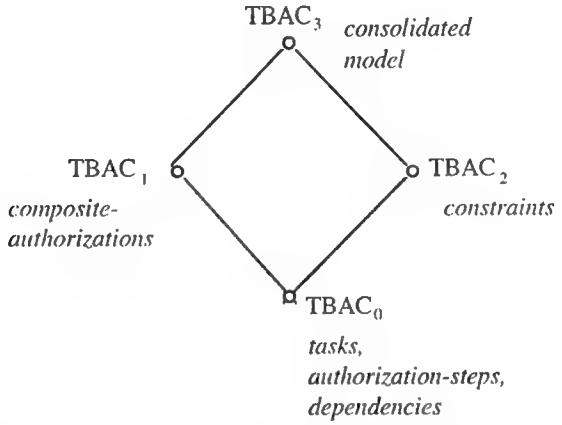


Figure 8. A framework for a hierarchy of TBAC models

Figure 8 shows our framework. $TBAC_0$ is a base model and is thus at the bottom of the lattice. It provides some basic facilities to model tasks, authorization-steps, and dependencies relating various authorization-steps. $TBAC_0$ is a very general and flexible model and is thus the minimum requirement for any system incorporating task-based authorizations. The advanced models $TBAC_1$ and $TBAC_2$ include (inherit) $TBAC_0$ but add more features. $TBAC_1$ incorporates the notion of composite authorizations (discussed in Section 3.1.3) whereas $TBAC_2$ adds constraints. Finally, $TBAC_3$ is the consolidated model that includes $TBAC_1$ and $TBAC_2$ and by transitivity $TBAC_0$.

Formulating such a family of models has many benefits. Researchers and developers can compare their system implementation of TBAC concepts with this family of models. A family of models also gives developers the opportunity and flexibility to choose conformance points for their implementations and can thus serve as a guide and evolution path for additional features. We discuss each of these models in turn below.

3.1.2 The Model TBAC$_0$

We will now describe the model TBAC$_0$ in detail. We describe the various attributes or components that make up every authorization-step, followed by its life-cycle, and lastly the dependencies that are used to model authorization policies.

3.1.2.1 Components of an authorization-step

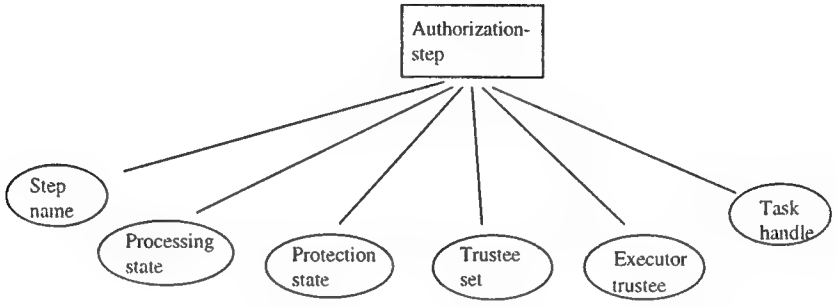


Figure 9. Basic components of an authorization-step

Every authorization-step has to specify a variety of components, as shown in Figure 9. We now describe briefly each of these components:

- *Step-name*: this is an identifier.
- *Processing-state*: The current processing state indicates how far the authorization-step has progressed in its life-cycle (discussed in Section 3.1.2.2).
- *Protection-state*: The protection-state defines all potential active permissions that can be checked-in by the authorization-step. The current value of the protection-state, at any given time, gives a snapshot of the active permissions at the time. Associated with every permission is a validity-and-usage specification. The validity-and-usage-specification specifies the validity and usage aspects of the permissions associated with an authorization-step. It will thus specify how the use of the permissions will relate to the authorization remaining valid (or becoming invalid).
- *Trustee-set*: This contains relevant information about the set of trustees that can potentially grant/invoke the authorization-step such as their user-identities and roles.
- *Executor-trustee*: This records the member of the trustee-set that eventually grants the authorization-step.
- *Task-handle*: This stores relevant information such as the task and the event identifiers of the task from which the authorization-step is invoked.

Let us now formalize these concepts.

Definition 1. We define a permission, p, as a tuple (s,o,a,u,as) where s stands for the subject or trustee and o represents an object for which the subject is given the right to perform action a u times within an authorization-step instance as. A permission is

always associated with an authorization-step instance and its associated protection state. If P is the set of permissions, then

$$P \subseteq S \times O \times A \times U \times AS$$

where S is a set of subjects/trustees

O is a set of objects

A is a set of action names

U is the usage and validity specification; a non-zero integer indicating the number of uses left (the special symbol ∞ is used to indicate unlimited uses) and a flag v indicating if the last use will make the authorization-step invalid.

AS is the set of authorization-step instances.

Definition 2. For each authorization-step instance, *as*, there is an associated protection-state SS_{as} defined by

$$SS: AS \rightarrow 2^P$$

$$SS_{as} = \{(s,o,a,u,as') \in P \mid as' = as\}$$

Definition 3. Each authorization-step instance, *as*, has a name. The following maps define additional attributes for each authorization-step.

Current Processing-state,	CPS : AS $\rightarrow$ PS, where PS is a set of all processing states
Protection-state,	SS : AS $\rightarrow 2^P$
Trustee-set,	TS : AS $\rightarrow 2^S$
Executor-trustee	ET : AS $\rightarrow$ S, $ET_{as} \in TS_{as}$
Task-handle	TH : AS $\rightarrow$ T, where T is a set of tasks

We also state informally two properties:

Property 1. Executor Assignment. For every authorization-step, *as*, the executor-trustee (ET) component is null until *as* transitions into the "started" processing state.

Property 2. Non-replaceable Executor. Once an executor trustee is assigned to an authorization-step, it is fixed for the entire lifetime of the step.

Property 3. Disjoint Protection States. The protection states associated with various authorization-steps are disjoint; as a result, every authorization-step instance has a unique protection state. Thus, given a set of authorization-steps $a_1, a_2, \ldots a_k$, and their respective

protection states, p_1, p_2, ...p_k, the intersection of two or more of these states will be empty. Formally, for any p_i and p_j, i is not equal to j, $p_i \cap p_j = \phi$.

3.1.2.2 Processing states and life-cycle of authorizations

As mentioned earlier, an authorization is not static; rather, it has a lifetime and a life-cycle associated with it. To better understand the execution aspects of authorizations, it is useful to consider the processing states that every instance of an authorization-step goes through during its life-cycle.

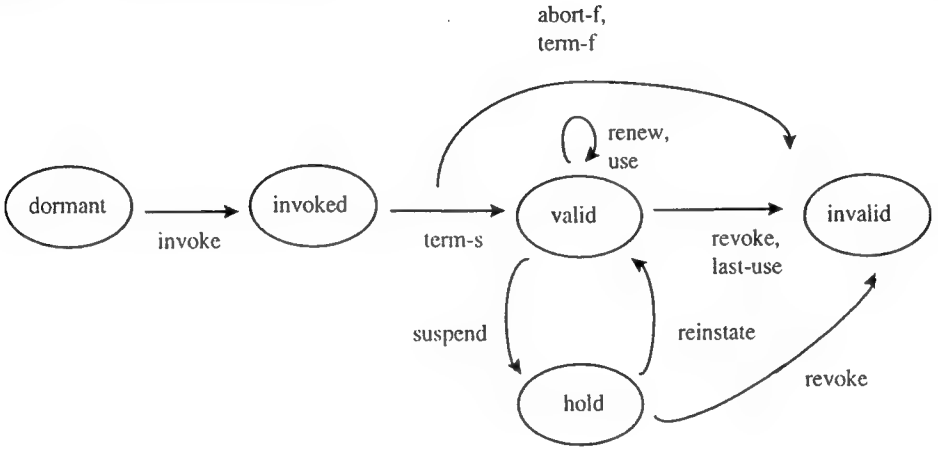


Figure 10. The basic life-cycle of an authorization-step

A simple view of this life-cycle is to consider every authorization-step instance as going through five states, namely dormant, invoked, valid, invalid, and hold, as shown in Figure 10. An authorization is dormant when it has not been invoked (requested) by any task. Once invoked, an authorization-step comes into existence, and will be processed. If this processing is successful, the authorization-step enters the valid state. Otherwise, it becomes invalid. In the valid state, all associated permissions with the authorization are activated and thus available for consumption. From the valid state, an authorization-step will undergo further processing and eventually reach the end of its lifetime and enter the invalid state. Furthermore, a valid authorization-step may be put on hold temporarily. When this happens, all permissions associated with the authorization-step are inactive and cannot be used to gain any access until this hold is released and the validity reinstated. Eventually, when an authorization becomes invalid, it ceases to exist, and is deleted from the system.

To get a more detailed description of what happens to an authorization during its lifetime, one can derive a more elaborate state diagram such as that shown in Figure 11. This more elaborate state diagram recognizes the dimension of use of permissions. A permission that is in the protection state of an authorization-step is consumed if any action that is enabled by the authorization-step requires the permission. Every operation request thus decrements the use count of the permission. Once the use limit is reached an action will no longer succeed as TBAC ensures that the required permission is no longer available.

Figure 11 is a direct refinement of Figure 10. The aborted and started states of Figure 11 are a refinement of the invoked state of Figure 10. Similarly, the valid, hold and invalid states of Figure 10 are each refined into a pair of corresponding used and unused states in Figure 11.

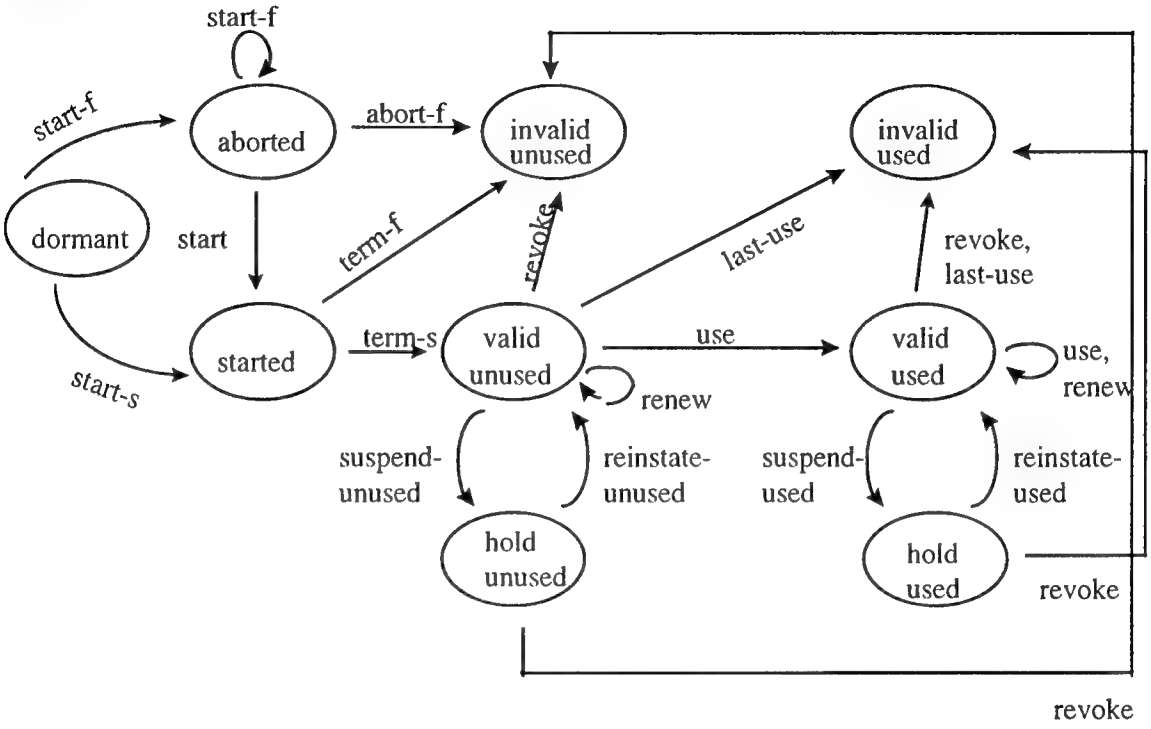


Figure 11. Detailed processing states of an authorization-step

We describe each of the processing states below.

- ***Dormant***: An authorization-step is in this state if it has not been invoked by any task. Equivalently, the dormant state can be viewed as one where the authorization-step does not as yet exist. In particular, the protection state of the authorization-step is empty.
- ***Started***: Once an authorization-step has been successfully invoked, it enters this state where processing begins.
- ***Aborted***: The aborted state is in many ways similar to dormant except that a failed attempt to start the authorization-step was made in this case.
- ***Valid-unused***: Once an authorization-step has been started, subsequent successful processing will make it transition into the valid-unused state.
- ***Valid-used***: If an authorization was in a valid-unused state, and it is subsequently used or consumed, then it enters the valid-used state. Depending on policy, an authorization may be used multiple times before it enters the invalid state.

- ***Invalid-unused***: This state is entered if certain conditions for an authorization to be valid are not met upon termination or if the authorization had entered the valid-unused state and was subsequently revoked.
- ***Invalid-used***: This state is entered either as a result of a last-use transition from the valid-unused state or as a result of a revoke or last-use event (transition) from the valid-used state.
- ***Hold-unused:*** In this state, the unused authorization is temporarily suspended. All associated permissions will thus be inactive.
- ***Hold-used:*** The authorization is temporarily suspended. All associated permissions will thus be inactive.

We can explain some of the semantics associated with the various states and transitions by considering the sample authorization-step below.

authorize_prepare_voucher {clerk}

In this example, an authorization to prepare a voucher is requested by a user in the role of a clerk. When this step is invoked and an instance of this authorization-step is created, a type-check is made to ensure that the "prepare" permission is allowed between the voucher and clerk type. If this check succeeds, the step transitions into the started state and the executor-permissions are checked-in (activated) into the protection-state. Between the started and valid-unused or invalid-unused states, there are no changes in the protection state. Once the step reaches the valid-unused state, the executor- permissions are checked out and the enabled-permissions are checked into the protection state.

These enabled-permissions will allow other actions to continue in the overall workflow. At some point, the authorization-step will become invalid and any remaining permissions in the enabled-permissions set will be checked out (deactivated).

3.1.2.3 Basic dependencies to construct authorization policies

In the previous sections, we discussed authorization-steps. However, in any application or workflow logic, authorization steps do not stand in isolation. Rather, security policy often requires dependencies between them. We now discuss various dependencies and constructs that relate authorization-steps to each other and constrain their execution and behavior. These dependencies can thus be used to formulate enterprise-oriented authorization policies. Such policies will mandate the propagation of authorizations from one employee to another. This is captured neatly by specifying (as policy) the authorization-steps that are part of enterprise workflows.

We specify dependencies in terms of existential, temporal, and concurrency relationships that hold between events (or states resulting from the occurrence of events). Given an authorization-step A, we use the following notation for the various states of A:

- A^d : the dormant state
- A^s : the started state

- A^{a} : the aborted state
- A^{v-} : the valid-unused state
- A^{v+} : the valid-used state
- A^{i-} : the invalid-unused state
- A^{i+} : the invalid-used state
- A^{h-}: the hold-unused state
- A^{h+}: the hold-used state

We list the dependency types and their meanings (interpretations) below:

1. $A1^{state1} \rightarrow A2^{state2}$: If A1 transitions into state1, then A2 **must** also transition into state2.
2. $A1^{state1} < A2^{state2}$: If both A1 and A2 transition into states state1 and state2 respectively, then A1's transition must occur **before** A2's
3. $A1^{state1} \# A2^{state2}$: A1 **cannot** be in state 1 concurrently when A2 is in state2.
4. $A1^{state1} \Vdash A2^{state2}$: A1 **must** be in state1 whenever A2 is in state2.

The first two dependency types $\rightarrow$ and $<$ express existential and temporal predicates and as such are best interpreted as predicates between transition events that lead to changes in the processing states of authorization-steps. They were originally proposed by Klein [8] to capture the semantics of database transaction protocols. The other dependencies express concurrency properties.

In a later subsection, we will illustrate the use of these dependencies with an order-processing example. Let us now formalize the concept of dependencies:

Definition 4. We define a dependency type as one of the following: $\rightarrow$, $<$, #, or $\Vdash$. We define DT, the set of dependency types, as $\{\rightarrow, <, \#, \Vdash\}$.

Definition 5. We define a dependency instance *d* as a tuple (*a1, dt, a2*) for which an assignment relation holds from a1 to a2. If D is the set of dependencies, then

$$D \subseteq AS \times DT \times AS.$$

3.1.2.4 Formal characterization of $TBAC_0$

We now formally define model $TBAC_0$ as follows.

Definition 6. The $TBAC_0$ model consists of the following:

- AS, a set of authorization steps;
- SS, a set of protection states;

- P, a set of permissions;
- D, a set of dependency instances;
- astep: SS → AS, a function mapping each protection-state to a single authorization-step; and
- pstate: AS → SS, a function mapping each authorization-step to a single protection state.

3.1.2.5 An order-processing example

Figure 12 shows a typical order-taking workflow scenario in a data-flow-like notation. The circles represent sub-tasks (sub-processes) within an overall order-taking task. Each subtask uses one or more documents/records as input and, in turn, may produce other documents as outputs. The various authorization-steps at the various sub-tasks are underlined.

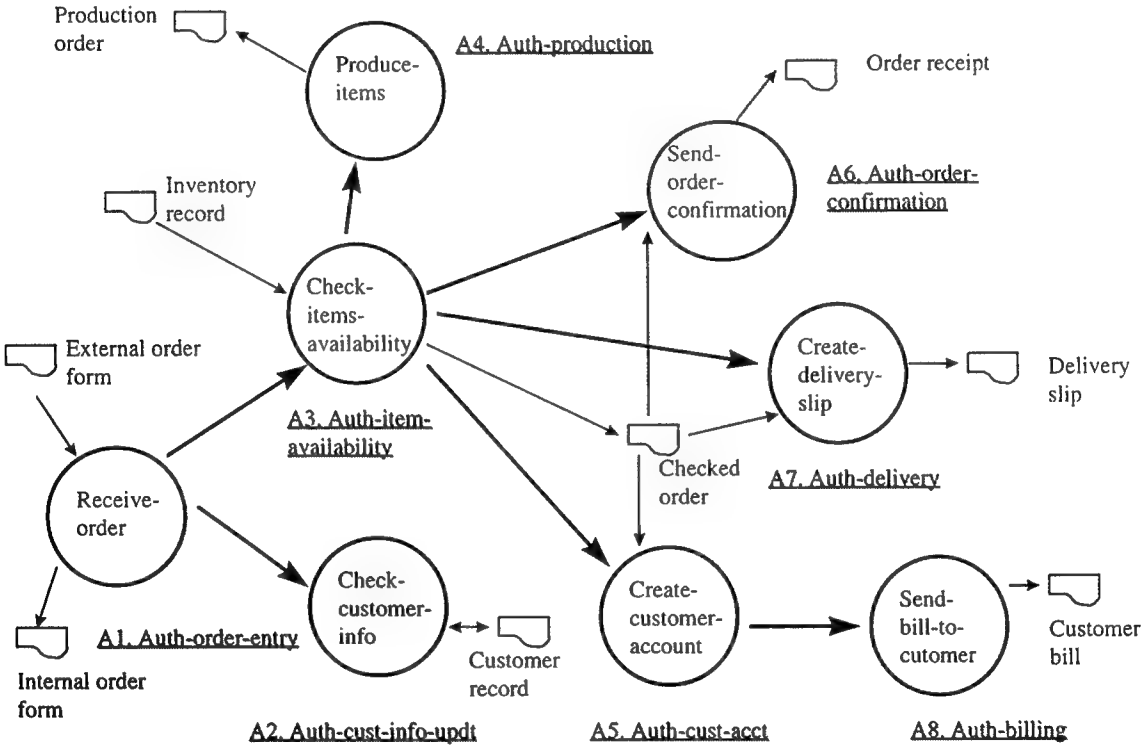


Figure 12. An order processing example

A typical order cycle is initiated at a merchant by the arrival of an external order from a customer and this is processed by the Receive-order sub-task. The Check-customer-info sub-task is then invoked to check if data on the order form is consistent with the customer's existing record and the latter is updated if necessary (such as when the address is out of date). Next, we check if the number of ordered items are available in stock. If this is not the case, the Produce-items sub-task is invoked to request the production of more items. On the other hand, if the inventory is adequate to meet the

order, a customer account is created for this order, and an order confirmation is sent to the customer. Next, a delivery slip is created to accompany the delivery and finally the customer is billed for the order.

We illustrate below the authorization-steps and dependencies in the above order-taking workflow application. There are eight authorization-steps:

- A1: auth-order-entry. This step activates the required permissions to file the external order, to create an internal order from the external order, and to write into the internal order.
- A2: auth-cust-info-updt. Involves the permissions required to update the existing customer record.
- A3: auth-item-availability. This step should succeed only if the inventory (quantity on hand) is adequate to satisfy the order. This requirement forms a start condition for the step. For a discussion on start conditions, see Section 3.1.4.
- A4: auth-production. This is an authorization for the production of more items. It gives permissions to create and to write a production order.
- A5: auth-cust-acct. This authorizes creating and writing a new customer account for an order.
- A6: auth-order-confirm. This authorizes an order receipt to be created and sent to the customer, thereby confirming the order.
- A7: auth-delivery. This authorizes delivery of the items and includes the permissions to create, write, and mail the delivery slip.
- A8: auth-billing. This authorizes billing the customer and activates permissions to create, write, and mail the bill.

The table below summarizes the values of the various components of each authorization-step. For brevity, we do not show the task-handles (TH). Permissions that are part of the protection-state are named by labeling them with the prefix "p" followed by the object to which the permission applies. We also use some notations to specify the validity and usage characteristics of the protection state. These determine how many times the permissions in the protection state can be used, and how this use can make the authorization-step invalid.

To specify usage, we attach in brackets the number of times the permission can be used. Thus,

p-read-int-order (n)	specifies that the internal order can be read at the most n times during the lifetime of the corresponding authorization-step.

We attach validity specifications as follows;

p-read-int-order (n,v) specifies that the authorization-step becomes invalid after n usages of this permission.

Table 1. Sales order authorization steps and their components

	STEP-NAME (SN)	TRUSTEE-SET (TS)	EXECUTOR-TRUSTEE	EXECUTOR PERMS (EP)	ENABLED-PERMS (SS)
A1.	**auth-order-entry**	order-entry-clerk	Tom	p-read-ext-order	p-file-ext-order (1) p-create-int-order (1) p-write-int-order (1,v)
A2.	**auth-cust-info-updt**	account-clerk	Smith	p-read-cust-rec	p-write-cust-rec (1,v)
A3.	**auth-item-availability**	inventory-clerk	Bob	p-read-qty-on-hand	p-debit-qty-on-hand (1) p-create-checked-order (1) p-write-checked-order (1,v)
A4.	**auth-production**	prod-manager	Anne	p-read-inventory-rec	p-create-prod-order (1) p-write-prod-order (1,v)
A5.	**auth-cust-acct**	account-clerk	Krista	p-read-checked-order	p-create-cust-acct (1) p-write-cust-acct (1,v)
A6.	**auth-order-confirm**	order-confirm-clerk	Bill	p-read-checked-order	p-create-order-receipt (1) p-write-order-receipt (1) p-mail-order-receipt (1,v)
A7.	**auth-delivery**	shipping-clerk	John	p-read-checked-order	p-create-delivery-slip (1) p-write-delivery-slip (1) p-mail-delivery-slip (1,v)
A8.	**auth-billing**	billing-clerk	Mary	p-read-cust-acct	p-create-cust-bill (1) p-write-cust-bill (1) p-mail-cust-bill (1,v)

In the examples in the above table, we have specified that all permissions in the various protection states are one-time permissions. In general, this may not be the case (for example, we may allow multiple reads or browsing, but only a single write or update). It is also important to note that the subjects in the enabled-permissions are different from executor-trustee.

To elaborate on the entries in this table, consider the first authorization-step A1, auth-order-entry. The entries in the table say that authorization A1 is to be granted by a trustee (user) in the role of an order-entry clerk. To process the authorization, the order-entry-clerk would need read permission for the external order form. Once the clerk grants the authorization, he activates (checks-in) permissions to file (archive) the external-order, to create a new internal order, and to write into the internal order. These permissions are

now available to other designated subjects in the system. Once the write permission is used (as a result of the internal order being filled out), the authorization becomes invalid. The other authorization-steps can be similarly elaborated.

Given the above authorization-steps, one can infer some sample policy governing the authorizations. Some of the policy requirements include the following:

- Authorization to update customer record can be granted only after receiving the successful authorization for order-entry. This can be expressed by the dependency

  $\text{Auth-order-entry}^{V+} < \text{Auth-cust-info-updt}^{S}$

- Authorization confirming the availability of the items has to be granted before authorizations for order-confirmations, delivery, creation of customer accounts, and billing. We thus have

  $\text{Auth-item-availability}^{i+} < \text{Auth-order-confirm}^{S}$

  $\text{Auth-item-availability}^{i+} < \text{Auth-cust-acct}^{S}$

  $\text{Auth-item-availability}^{i+} < \text{Auth-delivery}^{S}$

  $\text{Auth-item-availability}^{i+} < \text{Auth-billing}^{S}$

- If authorization for availability cannot be granted, then authorization for the production of more items **must** be granted. This can be specified by the existential dependency:

  $\text{Auth-item-availability}^{i-} \rightarrow \text{auth-production}^{S}$

- Authorization for order-confirmations, creation of customer accounts, delivery-slips, and billing cannot be valid when there is a valid authorization to produce more items.

  $\text{Auth-production}^{V- \text{ or } V+} \quad \# \quad \text{Auth-cust-acct}^{V- \text{ or } V+}$

  $\text{Auth-production}^{V- \text{ or } V+} \quad \# \quad \text{Auth-delivery}^{V- \text{ or } V+}$

  $\text{Auth-production}^{V- \text{ or } V+} \quad \# \quad \text{Auth-billing}^{V- \text{ or } V+}$

3.1.3 The model $TBAC_1$ to support composite authorizations

Having discussed $TBAC_0$, we next discuss $TBAC_1$ and $TBAC_2$. The model $TBAC_1$ supports the notion of composite authorizations. A *composite authorization* is an abstraction that encapsulates two or more authorization-steps. This is convenient when an authorization-step is too fine-grained a unit to express authorization requirements at a high (abstract) level.

For example, consider the authorization to transfer funds from one bank account to another. Such an action typically requires two authorizations. The first authorization is for withdrawal of funds from the source account and the second to deposit funds into the

target account. However, it is useful for modeling purposes to think of a more composite abstraction called "authorize-transfer" that consists of the individual authorization-steps.

Thus a composite-authorization consists of a set of component authorization-steps. These component authorization-steps can be related to other steps within the *same* composite-authorization through various dependencies. In other words, the authorization-steps of a composite-authorization are not visible externally to other authorization-steps outside the composite-authorization. The motivation for this restriction comes from a desire to follow sound software-engineering principles, especially those related to encapsulation and information hiding. Thus, to the external world, a composite-authorization is a single abstraction.

Collectively, the above properties and restrictions impose different semantics during the lifetime of a composite-authorization. In particular, we have to reexamine the notions of when we consider a composite-authorization to be started, valid, and invalid. We approach these issues by associating a *critical-set* of component authorization-steps with every composite-authorization. The critical-set is a subset of the component authorization-steps. We consider a composite-authorization to have started when any member of the critical-set has reached the started state. To be considered valid, all steps in the critical-set have to reach their respective valid states. On the other hand, a composite-authorization is considered invalid as soon as any step in the critical-set becomes invalid.

In addition to the validity associated with the critical-set, a composite-authorization may declare other non-critical-sets of authorization-steps to capture additional states of validity. However, these other sets can become valid only when the critical-set itself is valid and can remain valid only as long as the critical-set remains valid. Collectively, the critical-set along with the non-critical sets, define progressive states (checkpoints) of validity. The specification of a critical-set within a composite-authorization should thus be done with careful thought to some minimal notion of validity that ensures consistency with authorization policies for the enterprise.

3.1.4 The model $TBAC_2$ and constraints

As mentioned earlier, $TBAC_2$ supports more advanced notions of constraints. Thus $TBAC_2$ would be more suitable for an organization that finds $TBAC_0$ to be too open-ended or not having tight enough controls.

We classify constraints as static or dynamic constraints. Static constraints are those that can be defined and enforced when authorization-steps are specified. Dynamic constraints, on the other hand, are those that can be evaluated only at runtime, as authorization-steps are being processed.

In $TBAC_2$, the basic structure of an authorization-step has two components in addition to those present in $TBAC_0$. We describe these below:

- **Start-condition (SC).** This component can be used to specify a rich set of constraints that govern whether an authorization-step can make a transition into the started state. We are currently investigating both static and dynamic start conditions.
- **Scope (SP).** This component controls the visibility of an authorization-step with respect to other authorization-steps when formulating and enforcing authorization policies. Thus scope can be used to control if an authorization-step is visible to an entire workflow, a task, or other finer units such as sub-tasks.

We are currently investigating other static constraints for authorization-steps such as:

- *Constraints on processing state*: This constraint can be used to remove certain processing states (such as hold) from the life-cycle of a step.
- *Constraints on protection state*: This can be used to constrain the permissions that are allowed in the protection state (i.e. activated by the step).
- *Constraints on trustee-set*: This can be used to constrain the type as well as the instances of the trustees that can belong to this set. For example, we may want to constrain that the trustee be of type role and limited to instances of project managers and supervisors.
- *Constraints on executor permissions*: This can be used to specify what permissions are not allowed to be among the executor permissions.

The most obvious examples of dynamic constraints are those involving dynamic separation of duties/roles and coincidence of roles. Consider the following four authorizations (for brevity we show only the step name and the trustee-name specified in terms of roles).

A1: auth_prepare_check {clerk}

A2: auth_approve_check {supervisor}

A3: auth_issue_check {clerk}

A4: auth_reapprove_check {supervisor}

To prevent fraud and implement various checks and balances, the enterprise policy may dictate that the clerks performing steps A1 and A3 be distinct (separation of duties) while the supervisors for steps A2 and A4 may be the same. However, since any clerk or supervisor in the enterprise may be allowed to first perform A1 and A2 respectively, these constraints can be evaluated only at runtime. $TBAC_2$ allows for the specification of such dynamic constraints. These dynamic constraints are evaluated by looking at the history of the executor trustees in the authorizations that have been invoked. $TBAC_2$ also allows considerable modeling flexibility by allowing the reach of such dynamic constraints to be influenced by other static constraints such as scope. Thus we may specify that a dynamic separation of duties requirement holds across the scope of a sub-task, task, or other coarser unit. By keeping track of the executor trustees of invoked authorizations and combining the notions of dependencies and scope, the $TBAC_2$ model

can be used to provide a much more powerful and general approach to specifying separation of duties requirements than transaction control expressions (proposed in [13]).

3.2 Visual languages and Authorization Modeling

One of the novel research directions that we are pursuing in this project is the use of visual languages [6] to specify authorization-steps and authorization policies. We start our discussion on this aspect of our research by giving a brief introduction to visual languages.

A visual language allows us to construct pictorial representations of conceptual entities and operations. A visual language environment is thus a tool through which one can compose iconic, or visual, sentences that convey meaning. Users express their requests as visual sentences that consist of spatially organized icons on a screen. This is in contrast to typical graphical user interfaces (GUIs) that provide a limited set of icons with predefined meanings and a restricted set of iconic commands.

The visual language research for our project is being undertaken in collaboration with researchers the Universities of Pittsburgh and Salerno. These researchers have jointly developed the Pittsburgh-Salerno Iconic System (PSIS) [6], which lets users design, specify, and interpret custom visual languages for different applications. The system consists of two major subsystems. The first is the visual-language compiler, which lets the user input, translate, and execute visual sentences. The second subsystem is the visual-language generator, which, given user-supplied visual sentences, generates the grammar and the related semantic functions.

3.2.1 Motivation to use visual languages

Visual languages overcome the limitations of GUIs and allow users to visually reason and communicate. Users can program and query using visual icons and constructs that are intuitively meaningful, especially in relation to the metaphors and mental models with which the user is familiar. Tools based on visual languages reduce the time and expense required for user training. As well, communicating requests and ideas in an intuitive visual fashion reduces errors that users make with typical text-based interaction.

From the security standpoint, we are motivated to explore visual languages to exploit their advantages when applied to security modeling and security tools. Visual languages will allow us to

- To move away from system-centric approaches to security modeling and specification to a paradigm that is more enterprise and metaphor-oriented.
- To create enterpise-oriented, user-friendly security administration and monitoring tools.

As discussed earlier, the historical development of security models has focused on the protection of system-centric resources and abstractions. As is perhaps inevitable, many security modeling and administration tools reflect this view. Current approaches to security administration consist of setting access control permissions and other low-level

information. Our goal is to move towards a policy-based approach to security modeling and administration. However, when dealing with policy we must confront the issue of how to capture the intuitive meanings embedded in policy statements. This requirement makes visual languages attractive. Another advantage of visual languages is that it allows us to design interaction modes for end users that are aligned with the mental models and metaphors with which the user is familiar.

3.2.2 A two-tiered visual-language framework for TBA

Figure 13 shows our two-tiered framework for applying visual languages for task-based authorizations [7]. The bottom-tier approaches visual languages from the need to specify the various components of authorization-steps. Thus our goal is to seek an appropriate "specification metaphor." We have chosen the visual metaphor of a combination lock for this purpose. The version of the combination lock that we are considering consists of multiple wheels. Each wheel corresponds to one component of an authorization-step. A user is guided to specify an authorization-step by being asked to set each wheel of the combination lock. When all the wheels have been set, the authorization-step is fully specified.

The top tier in our framework utilizes appropriate metaphors and composite icons to convey the basic idea of an authorization along with other additional meanings (adjectives). We have chosen the forms-signature metaphor for this tier. Thus the main icon at this tier conveys the notion of a signature (along with the act of signing) on a paper form. Additional meanings (such as if an authorization is valid, invalid, used, or unused) are conveyed by secondary icons attached to the main icon.

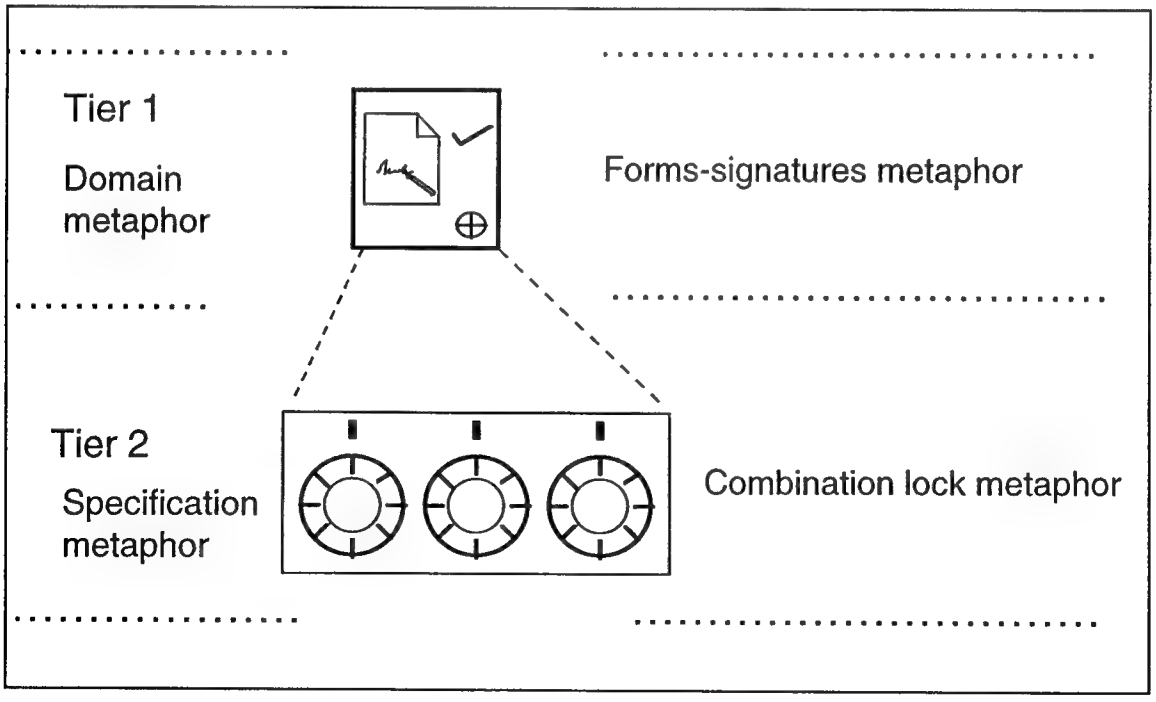


Figure 13. A two-tiered approach to using visual languages for TBA

3.2.3 Basic icons to visualize authorization steps and their meanings

We now discuss in more detail the icons for expressing authorization-steps and authorization policies. They are shown in Figure 14. The primary (main) icon is one that depicts an act of authorization (approval) and consists of the picture of a pen superimposed on the picture (icon) of a paper form  . The remaining icons are secondary icons and are used to convey additional meanings for an authorization when attached to a primary icon. The additional meanings are related to whether an authorization-step is valid, invalid, put on hold, unused, and used.

Icon	Meaning
	Authorization (approval) on a form
	Valid
	Invalid
	Valid and put on hold
	Unused
	Used

Figure 14. Basic icons

We may associate a combination of these additional meanings with an authorization-step. For example, an authorization-step may be both valid and unused. In this case, all the relevant secondary icons are attached to the primary icon. Figure 15 illustrates all combinations that are allowed (make sense) in our model. These correspond to the processing states of an authorization-step, discussed in an earlier section.

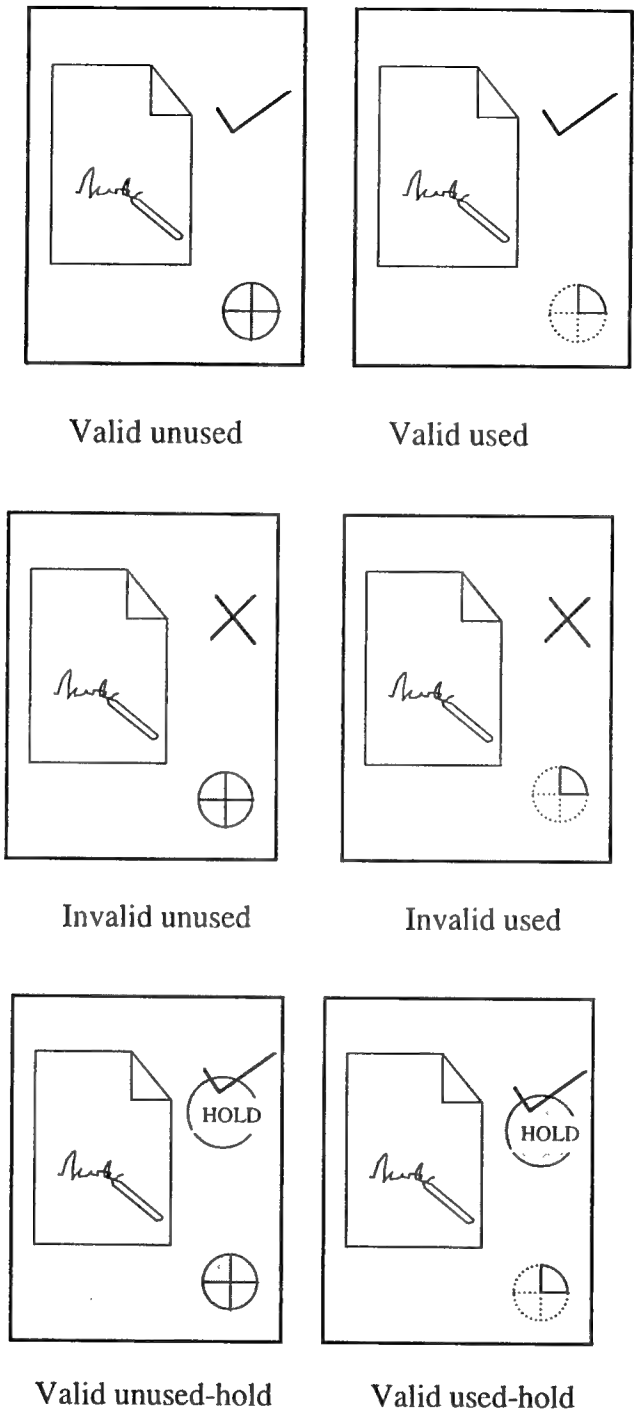


Figure 15. Conveying combinations of meanings through secondary icons

3.2.4 Visual sentences and authorization policies

Having discussed how we van visually specify an authorization-step and its states, we now turn our attention to the specification of authorization policies. Our goal is to provide a visual representation for each of the dependencies presented in Section 3.1.2.3. In other words, we should express them through visual iconic operators in a way that will

be intuitive to the end user. It is difficult to express the whole meaning of a dependency through a single icon. At the same time, the primary icons for individual authorization-steps should be part of the visual metaphor chosen for the dependencies. Basically, we should convey temporal and coexistence relationships among authorization steps. Thus the Signed Paper metaphor needs to be combined with other metaphors to express the dependencies. We have done this by using a combination of the "Calendar" and the "Color" metaphor, altogether with the "Signed paper form metaphor" as described below.

There is a picture representing a calendar, which can be seen as a composite icon of our language, and will be used to convey the temporal dependencies among authorization steps. We can imagine many different layouts for this icon. Let us consider a calendar formatted as a grid, where the icons for the single authorization steps are placed inside the cells of the grid, and each cell can host more than one icon. Thus, we will express authorization policies through iconic sentences with icons that can include the calendar icon and the primary icon for single authorization steps (signed paper forms).

The $A1^{state1} < A2^{state2}$ dependency rule can be represented through an iconic sentence having the calendar, and the icon for authorization step A1 placed on a calendar cell corresponding to a date prior to the one where we put the icon for A2. The states state1 and state2 according to the "Signed Paper Form" are encoded in the layouts of the icons for A1 and A2.

The two dependency rules '#" and 'lll', specified in Section 3.1.2.3, are implemented by inserting the two icons for A1 and A2 in the same calendar cell. We differentiate among them by using the "Color" metaphor. We color the calendar cell border with one of the two colors red and green for the "'#', and 'lll' dependencies, respectively. Alternatively, we can color the two icons for A1 and A2 both green for '#' and one green and the other red for 'lll'.

If we need to differentiate between the 'lll' and the '→' dependencies, we can add the '→' symbol between the two icons for A1 and A2.

We notice that it is possible to combine several dependency rules within one iconic sentence. For instance, we could have three icons on the same calendar foil to describe the three dependency rules "A1 < A2", "A1 < A3", and "A2 < A3" altogether. Consider another example: if A2 and A3 were put in the same calendar cell with the green border, we would express the dependency rules "A1 < A2", "A1 < A3", and "A2 lll A3", provided that A1 was in a calendar cell corresponding to a date that is prior to the one containing A2 and A3.

In order to control the consistency of these rules, we use an underlying syntax-driven mechanism that allows only feasible placements. Thus, if we have stated the rule "A1 < A2", the editor should not allow placing the icon for A2 in a date prior to the one for A1 on another calendar icon. We can use several calendar icons, in which the year can determine a temporal dependency (<) among the icons placed on different calendar icons.

The user is allowed to expand each single authorization placed in a calendar cell to access the Wheel locks based visual language, in order to set or modify some of the components for that authorization.

3.2.5 Syntactic Aspects of the visual language for TBA

The iconic sentences described in the previous sections need to obey to some syntactic rules, regarding not only the symbols that we use, but also their spatial relationships. In our first tier Visual language, a visual sentence is a spatial arrangement of authorization step icons, and operator icons that represent the dependency rules.

Several grammar models for visual languages have been developed. Among these, we focus on Positional Grammars and Relational Grammars.

In positional grammars, each icon represents a token, and it is characterized not only by its name, but also by its position in the 2D space. In our model, an icon carries many other attributes, but they are used only for semantic analysis. In traditional context-free grammars, the only possible spatial relation between two tokens was the horizontal concatenation. In positional grammar, we can define many types of spatial relations. Simple examples of spatial relations are Hor, which is the traditional horizontal concatenation, and Ver, which represents the vertical concatenation of symbols. More sophisticates relations can be Dia, which stands for diagonal concatenation, and Overlap, in which two icons can be placed in the same position in the 2D space. Thus, the parser needs information to decide which direction of the Two-dimensional space to move in order to scan the next symbol. A production rule of a Positional Grammar has a single non-terminal symbol on its left hand side, whereas on the right hand side it has a sequence of terminal and non-terminal symbols, separated by spatial relations. A positional sentential form of a Positional Grammar is an alternate sequence of symbols and spatial relations, where each symbol can either be a terminal or a non-terminal, that can be derived from the starting non-terminal symbol by applying some of the productions in the grammar. A sentence of the Visual language generated by a Positional Grammar is a Positional Sentential form where each symbol is a terminal, and hence it is an icon. To obtain the final layout of a sentence, the sentence needs a positional evaluation. In what follows, we give some simple examples of the definitions given above.

Suppose we have the following Positional Grammar (N, T, POS, S, P), where,

N = S, A, B, is the set of non-terminal symbols,

T = a, b, c, is the set of terminal symbols, which represent icons,

POS = Hor, Ver, Dia, is the set of spatial relations defined for this grammar,

S is the starting non-terminal symbol,

P is a set of productions. The productions we have defined for this grammar are:

1. S:= A Hor B
2. A:= a Ver A
3. A:= a
4. B:= b Dia c
5. B:= b

According to the definition we have given to the three spatial relations, the given grammar generates the following visual sentences:

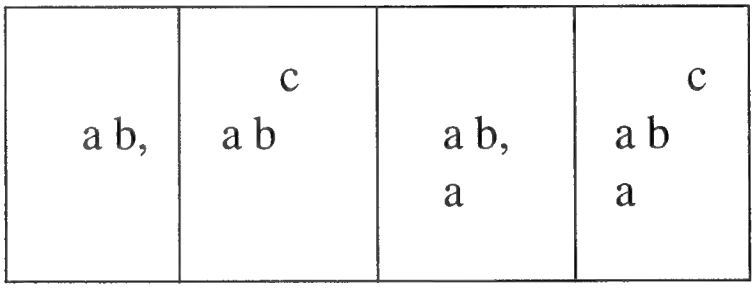


Figure 16. Visual sentenceses in positional grammar

Obviously, the number of Visual Sentences that can be generated with this grammar is infinite. In fact, we can have an infinite number of 'a' vertically arranged in each type of sentence. We have shown the sentences after they have been processed by the Positional Evaluator. In what follows, we show the derivation of the last Visual Sentence, from the symbol S to the final Positional Sentential form:

1 2 3 4
S ==> A Hor B ==> a Ver A Hor B ==> a Ver a Hor B ==> a Ver a Hor b Dia c

The last Positional Sentential form is also a sentence, since it contains only terminal symbols. From this, we derive the Visual Sentence by performing a positional evaluation.

Relation grammars present many similarities to Positional Grammars. The Positional Evaluation rules are replaced by relational predicates. The Production Rules are also similar, but the Spatial Relations are relational predicates.

For our Visual language, we will use Positional Grammars. We will provide a syntactic structure only for the first tier Visual Language. We could also provide a Positional Grammar for the Visual language, based on wheel locks. This would be very straightforward since the spatial arrangement of wheel locks does not convey any information. Thus, the grammar will reduce to a traditional string grammar with only horizontal concatenation.

Let us now define the Positional Grammar of the Visual language for TBA. Although we have provided several solutions to the problem, in what follows we have provided a grammar with very basic spatial relations. The grammar does not use the Overlap spatial relation.

In representing the calendar, we have different design choices. We will choose to represent it as a set of icons, each representing one cell of a grid. In the future, since we are considering implementing a more sophisticated set of spatial relations, we could reduce the whole calendar to one icon. We can enter new spatial relations in PSIS by defining the way they relate two symbols, by comparing their coordinates in the 2D space, according to a given formula. We will enter such formula in a library function.

Suppose we want to represent a calendar with the twelve months on the columns and up to thirty-one days on the rows. Actually, we may not be interested in this granularity of the calendar. What is important is to convey some temporal relationships. So, for simplicity, let us consider the calendar for the month of January only; we will need thirty-one icons for its visual representation. The layout of this calendar is sketched below:

MON	**1**	**8**	**15**	**22**	**29**
TUE	**2**	**9**	**16**	**23**	**30**
WED	**3**	**10**	**17**	**24**	**31**
THU	**4**	**11**	**18**	**25**	
FRI	**5**	**12**	**19**	**26**	
SAT	**6**	**13**	**20**	**27**	
SUN	**7**	**14**	**21**	**28**	

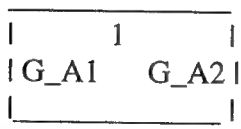
Figure 17. The calendar metaphor and its visual layout

To this set of icons, in which we could also include the dummy icons for the days of the week, we need to add the icons used for describing the single authorization steps. Since we will be using the color metaphor in conjunction with the Calendar and the Signed Paper form, each cell of the calendar will still be a composed icon. The latter can have just the number indicating the day, or in addition the icon for an authorization step, or two icons in the same cell with a colored cell's border to represent the dependencies. As said above, we can also use a combination of colored authorization steps, using the red, green, and yellow color.

The grammar productions for the example above are given in the following:

```
Auth_Policy   := Auth_Step | CAL Hor Auth_Policy |  CAL Ver Auth_Policy;
Auth_Step     := A1 | A2 .....|;
CAL           := Month Ver Grid;
Month         := January 1996 | .....;
Grid          := Week_col Hor Day_Grid ;
Week_Col      := Sun Ver Mon Ver .....Ver Sat;
Day_Grid      := Week1 Hor Week2 Hor Week3 Hor Week4 Hor Week5;
Week1         := Day1 Ver Day2 Ver.....Ver Day7;
.....
Week5         := Day29 Ver Day30 Ver Day31;
Day1          := Print1 Ver Auth_Dep | Print1;
.....
Print1        := blank Hor 1 Hor Blank;
Auth_Dep      := blank Hor Auth_Step Hor Blank |Can | Cannot | Must | Imply ;
Cannot        := Red_Step Hor blank Hor Red_Step;
Must          := Green_Step Hor blank Hor Green_Step;
Imply         := Green Step Hor => Hor Green_Step;
Red_Step      := Red_A1 | Red_A2 .....;
Green_Step    := Green_A1 | Green_A2 ....;
```

As for the blank non-terminal, since we have subdivided each calendar cell as a composed icon with six elementary icons, it will be a blank icon. Thus, for example, the 'III' dependency will be expressed through the following cell:



3.3 Prototype architecture and implementation

There are basically two broad objectives that have guided our development efforts in TBAC. The first is to model, from an enterprise perspective, various authorization policies that are relevant to organizational tasks and workflows. This requires a set of

user-friendly tools to help a security officer model and specify policies. Our second objective is to seek ways in which these modeled policies can be automatically enforced at runtime when the corresponding tasks are invoked.

To meet the above objectives, we developed two subsystems as part of our prototype.

1. *Policy editor.* This subsystem provides the tools and services required to interactively specify, modify, store, as well as retrieve authorization policies.
2. *Authorization server.* This is a server system that is capable of retrieving policies constructed by the policy editor and using such policies to provide runtime authorization enforcement.

In the next section, we discuss the high-level design and architecture of these subsystems.

3.3.1 High-level Design and Architecture

Figure 18 shows a high level architecture of our prototype. The shaded boxes indicate modules

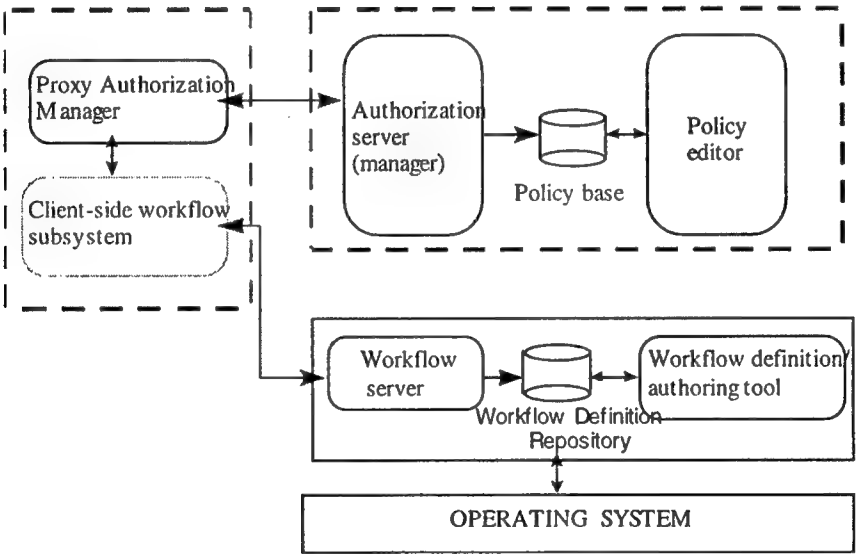


Figure 18. High level architecture of prototype implementation

that we are building while the unshaded boxes indicate commercial off-the-shelf (COTS) components. The philosophy behind the design of this architecture has been to not reinvent the wheel and to ensure that our development can be integrated with commercial systems. Hence we have not re-designed any workflow engine or a workflow management system (WFMS). Rather, we are building our authorization manager and policy editor on top of a commercial WFMS (Novell's Workflow extensions built on top of the Groupwise platform).

The basic idea is to use a policy editor to create and store the policies of individual applications (which are also called policies) in some persistent way in a policy base. When a client participating in a workflow requires an authorization, this request is communicated to a client-side proxy of the authorization manager. The proxy then sends the request over to the authorization manager (AM). The AM will then service the authorization request according to specified policy of the task and workflow from which the authorization originated.

3.3.2 The Policy Editor Subsystem

3.3.2.1 Design goals and framework

The policy editor is being built with the following design goals:

1. Ease of use.
2. Platform independence.
3. Support for distributed and nomadic computing.
4. Reusability beyond TBAC project.

Our policy editor framework caters to the typical enterprise setting. In particular, we assume that there will be a chief security officer or security administrator that is responsible for specifying and maintaining security policies using a policy editor. Our editor framework recognizes the following abstractions.

- Organization
- Organization-unit/Department
- Policy
- Policy-type
- Sentence
- Entity
- Relation
- Attribute

Let us elaborate on these further. As mentioned before, the term "policy" is used to refer to the security policy for an individual application or workflow type. However, such a policy always belongs to a policy-type. Examples of policy-types include authorization (such as for TBAC), audit, access control, and authentication. This is analogous to word processors that consider documents to belong to a type such as MSDOS-text, rtf, MS-Word etc. Every policy is made up of one or more sentences. A sentence models a single policy statement or expression. A sentence in turn consists of a triple of the form: (entity1, relation, entity2). Figure 19 and Figure 20 collectively show how these various abstractions are related.

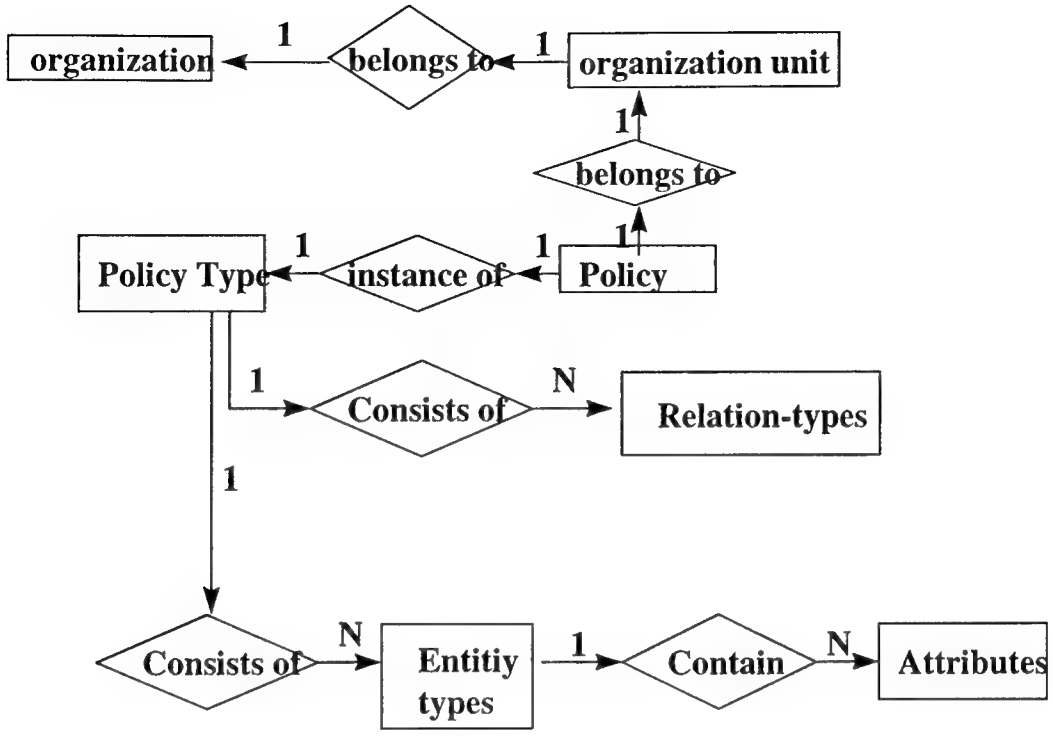


Figure 19. An ER diagram showing relationships between editor abstractions

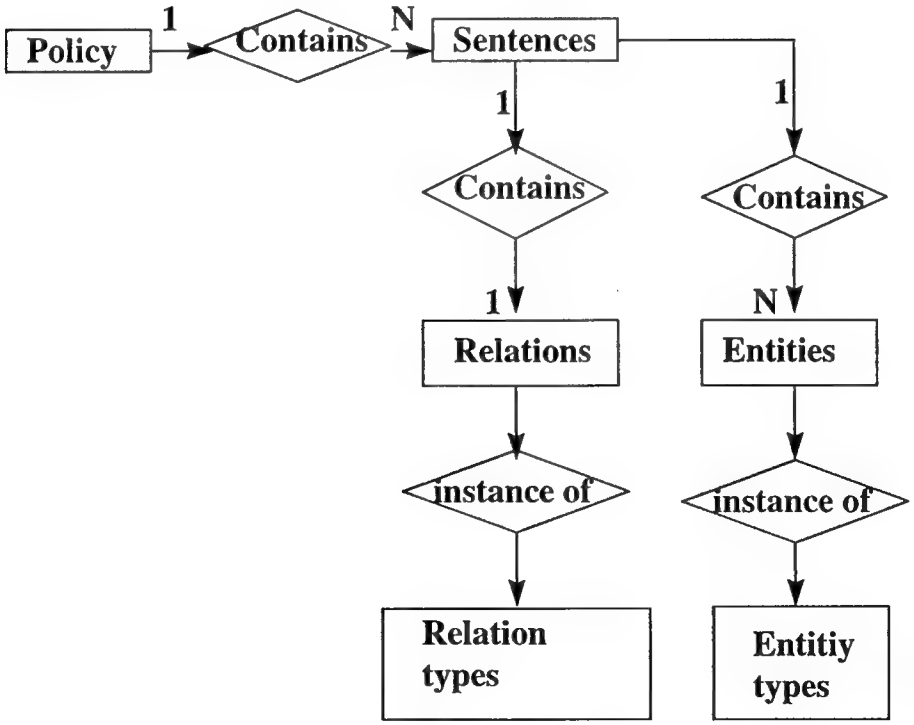


Figure 20. An ER diagram of the structure of a policy sentence

Figure 21 illustrates our software architecture for the policy editor. We use a three-tiered approach to policy representation and processing. Basically, a policy has three representations

- External Policy
- Intermediate Policy
- Internal Policy

We use the term *external policy* to collectively refer to the way a policy is represented externally, manipulated, and finally output by an editing tool. This will obviously depend on the user interface approach taken. In a text-based editor, this might be pure text tokens; in a graphical user, it might be graphical controls and icons; and in a visual language, it might be icons and iconic operators. The *internal policy* is a policy format that is used to store policies (policy sentences). Ideally, this would be a format that is readily amenable to efficient storage and query processing. For the TBAC project, our internal policy management is done using ODBC-compliant relational databases. Finally, the *intermediate policy* is a format that is independent of both the external and internal policy formats. The intermediate policy is used by the runtime authorization server and reasoning engine to enforce policies and to check the consistency of policy sentences. This kind of architecture provides the runtime server system with complete independence from editing tools and storage machines such as databases. This enables the latter to be supplied by third party vendors and increases the plug-and-play capabilities of the system.

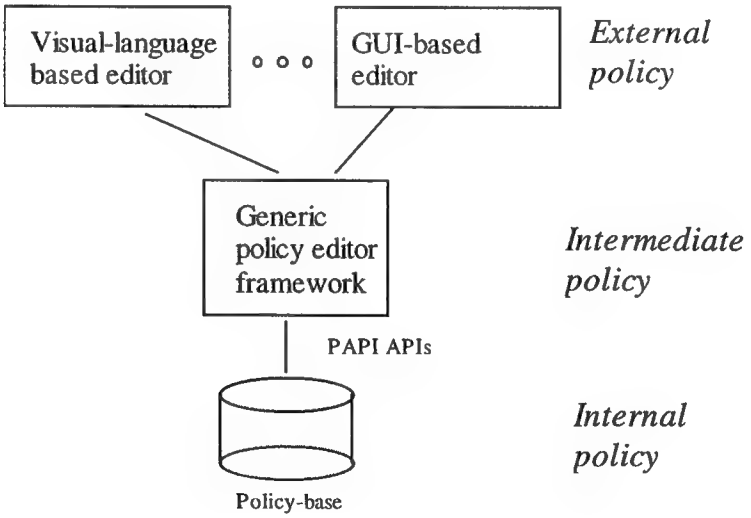


Figure 21. Policy Editor Architecture

3.3.2.2 Policy management APIs

Table 2 below summarizes the basic policy management APIs (PAPIs) that evolved during the course of the design of the policy editor. These APIs can collectively be used to define, store and retrieve authorization policies and are used internally to organize and implement the source code of the editor. They are designed to be general enough to accommodate other policy types, such those involving auditing, authentication, and access control. Our hope is that these APIs lead to a foundation for a general policy editor framework.

Table 2. Policy management APIs

API function name	**Parameter 1**	**Parameter 2**	**Parameter 3**
Get_all_depts_in_organization	in: org_name	out: dept_list	
Get_policies _in_dept	in: dept_name	out: policy_list	
Get_no_of_sentences	in: policy_id		
Get_sentence_id_list	in: policy_id	out: sentence_id_list	
Get_sentence	in: policy_id	in: sentence_id	out: sentence
Verify_sentence	in: policy_id	in: sentence_id	out: status
Save_sentence	in: policy_id	in: sentence	out: status
Open_policy	in: policy_id		
Save_policy	in: policy_id		
Define_meta_type	in: meta_type	in: type_name	
Get_policy_type	in: policy_name	out: policy_type	
Get_all_policy_types	in: org_name		
Get_all_policies_of_type	in: policy_type	out: policy_list	
Get_policy_entities	in: policy_type	out: entity_list	
Get_policy_relations	in: policy_type	out: relation_list	
Get_no_of_entities	in: policy_type	out: no_of_entities	
Get_no_of_relations	in: policy_type	out: no_of_relations	

3.3.3 The Authorization Server Subsystem

We now describe the authorization server subsystem in more detail.

3.3.3.1 High-level architecture

Figure 22 shows the high-level architecture of the authorization server. The authorization server module is not a single monolithic unit of software. It consists of

- One **Authorization Manager** (AM)

- Several **Case Managers** (CM)
- Several **Step Managers** (SM)

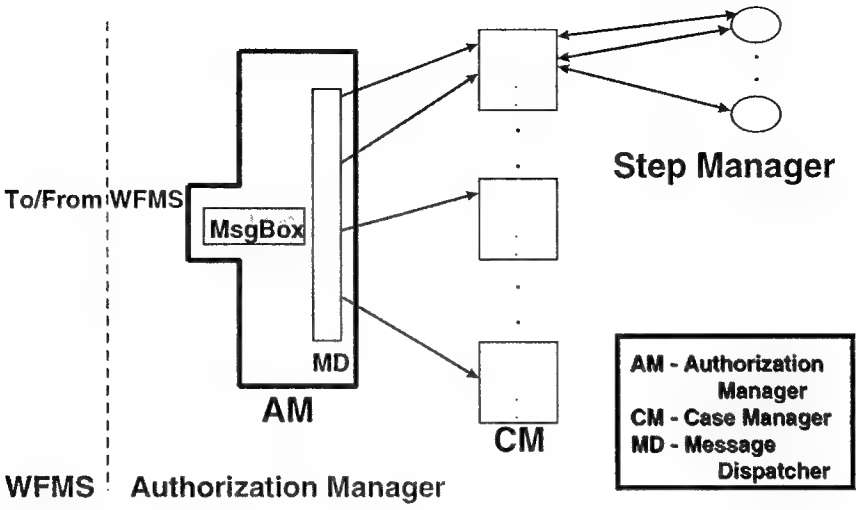


Figure 22. High-level architecture of the authorization server

The AM is more of a passive element in the whole runtime architecture. It is responsible only for routing messages from the WFMS to the various CMs. Whenever a message is received by the AM, it creates a request handler (with all requisite parameters) for that particular message. This request handler then communicates with individual CMs (if they already exist) or creates CMs (if need be) and then communicates with them. Each such request handler is a separate thread within the AM. These individual threads are also responsible for sending responses (from the CMs) to the WFMS. So, basically, the AM is an overseer or coordinator process in the whole runtime module.

Each CM is responsible for the management of the authorizations that are invoked from individual cases (instances) of a workflow type (class). This is to say that in our scheme, we envision that each workflow instance will have a separate set of authorization steps to be processed during the course of its lifetime. Such a complete set of authorization steps is managed by each CM. So, whenever a workflow instance invokes any authorization step for the very first time, a unique CM is created by a request handler thread (from the AM) to handle all possible authorization steps within that workflow instance. If such a CM already exists, as a result of a previous invocation of an authorization-step within the same workflow instance, the request for an invocation of an authorization step is simply forwarded to that particular CM. The CM, on receipt of a message from the AM, creates a request handler thread to handle that particular message. This thread then communicates with a particular SM or, if need be, creates a particular SM and then communicates with it.

The SM is the unit responsible for maintaining the state and progress of a particular authorization-step instance within a single workflow instance. That is, it is the SM that

keeps track of the various components of a particular authorization step (for example, current processing state, and use and validity counts). Each SM receives messages from a request handler thread of a CM, evaluates certain rules before it performs the request, and sends back responses to the thread regarding success or failure of the request.

In summary, we can say that the entire authorization module has a single global authorization manager that can, at any instance of time, manage several case managers, each of which, at any instance, can manage a number of step managers.

3.3.3.2 Client-server communications

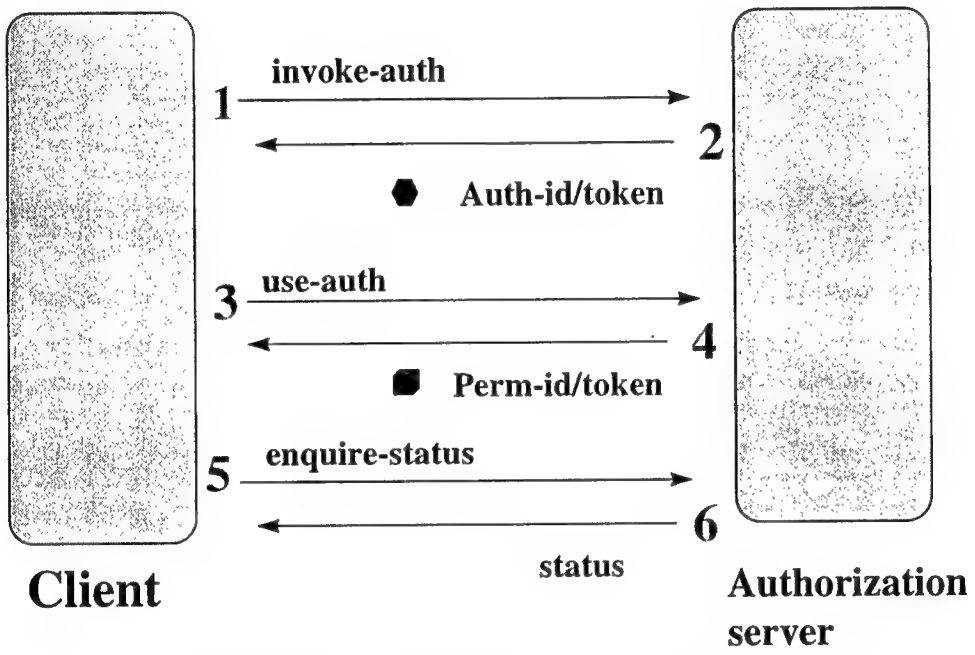


Figure 23. Messages for client-server communication

Figure 23 shows the basic six messages that can be exchanged between a client and the authorization server. For clarity, we do not show denials of requests or negative acknowledgments. These numbered messages are discussed below.

1. **Invoke-auth**. This message is sent from the client to the server when the client wishes to request an authorization. If the authorization can be granted, the server returns a reply, which forms the second message.
2. **Auth-id**. This second message is sent from the server back to a client if the client's previous request for an authorization can be granted. The server returns an authorization identifier or token that will have to be presented in subsequent requests by clients when they want to use the permissions in the authorization.
3. **Use-auth**. This message is sent from a client and used to present the authorization identifier and a request to use a permission.

4. **Perm-id**. This is the message sent back from the server in response to a use-auth message from a client.
5. **Enquire-status**. This message can be used by a client to enquire about the status of an authorization (this feature has not been fully implemented).
6. **Status**. This is a reply to an enquire-status message and contains details about the status of an authorization.

3.3.4 Software Documentation

In this subsection we discuss the configuration and management of the prototype as well as the overall organization of the various software elements used to build the prototype.

3.3.4.1 Software requirements

The project has used the following software packages and development environments.

1. *Object-oriented software development environment.* We used the Visual C++ development tool to implement the authorization manager.
2. *Java software development environment.* This is required to build a policy editor that will be web-based and platform independent. We have used Symantec's Visual Café interactive Java development environment to build the policy editor as a Java applet.
3. *Workflow management system (WFMS).* The WFMS is used to develop some workflow scenarios that involve authorization requests. We have utilized Novell's Groupwise Workflow platform for this purpose.
4. *ODBC-compliant Relational database management System (DBMS).* The DBMS is used by the policy editor and authorization server to store and to retrieve authorization policies (policies). We have used Microsoft's SQL Server version 6.5 for this.

3.3.4.2 Location and organization of files

The source and binary files for the policy editor and authorization server are organized into a directory called "Tba" as shown below in Figure 24.

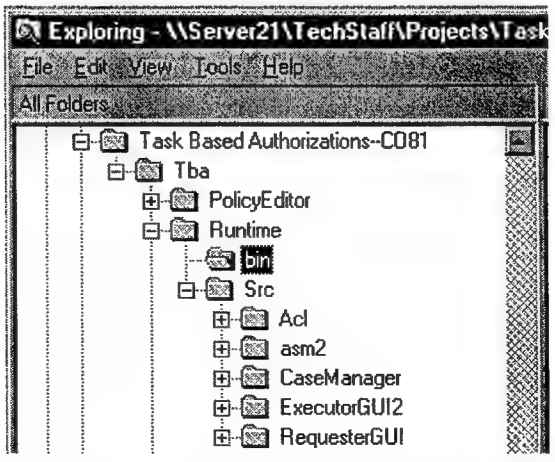


Figure 24. Directory/Folder organization of files

Table 3 summarizes the contents of each folder.

Table 3. Folders and their contents

Folder Name	Description of Files
Tba	This is the top folder
PolicyEditor	This folder contains all code and JAVA classes for the editor.
Runtime	This folder contains subfolders that hold the binary and source code for the authorization server.
Binary	This folder contains the executable files for the authorization server, case manager and the executor and requestor interfaces for communicating with the server.
Acl	Source code for manipulating NT Access control lists
Asm2	Source code for the authorization server
CaseManager	Source code for the case manager
ExecutorGUI	Source code for the GUI interface for executors of authorization steps.
RequestorGUI	Source code for the GUI interface for requestors of authorization steps.

3.3.4.3 Using the policy editor

In this section we summarize the main features of the policy editor. As mentioned previously, the policy editor is built in the JAVA language to enable platform independence. As such it can be run as a JAVA applet. The main features the editor provides is the ability to create, edit, and delete authorization policies. We will now briefly describe these features.

Invoking the Policy Editor

The editor can be invoked through the applet viewer facility in Visual café or through a standard web browser supporting JAVA applets. Once invoked, the editor displays the main (opening screen) shown in Figure 25.

The editor provides the means for an end-user to interact with the backend policy repository. The user can

- Create a new policy,
- Edit an existing policy, and
- Delete an existing policy.

Recall that each policy belongs to a particular policy type. The policy types are defined a-priori to the system. Therefore, through the editor users can create, modify or delete policies belonging to certain pre-defined policy types.

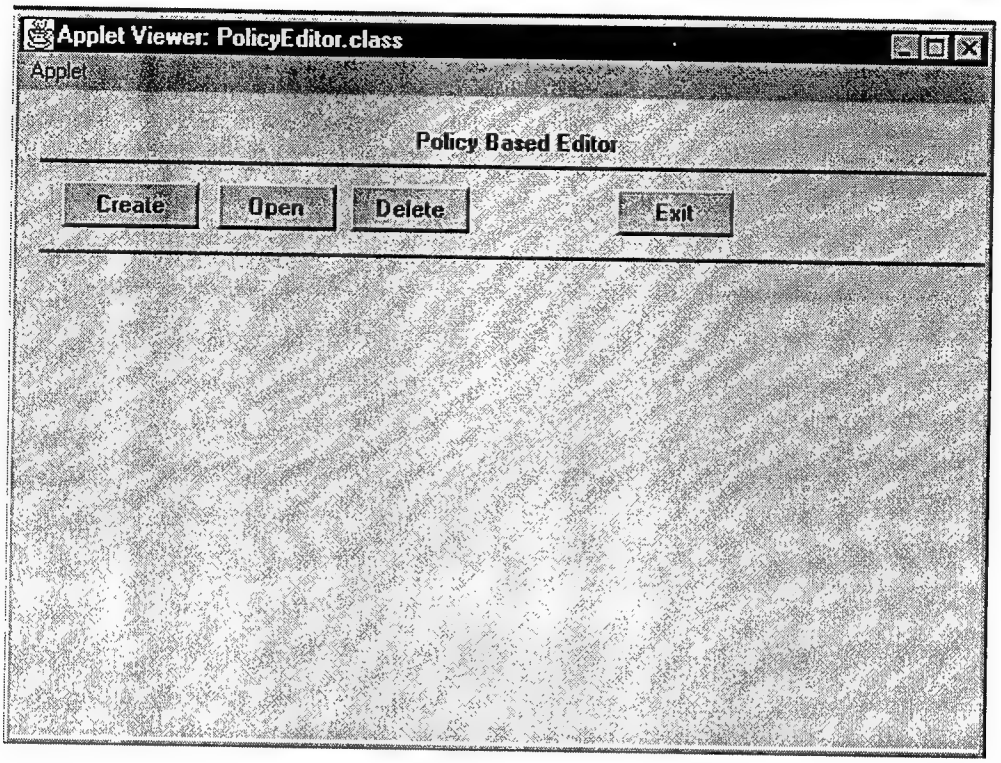


Figure 25. The opening screen of the editor

Each policy contains entities and sentences. The editor provides an easy and intuitive way in which an end user can create, edit or delete entities and sentences of a particular policy.

The initial screen of the applet (depicted in Figure 1) indicates the options available to the user. At this point ,the user can either

- Click on the CREATE button to start creating a new policy, or
- Click on the OPEN button to open, browse, and edit an existing policy from the database, or
- Click on the DELETE button to delete an existing policy, or
- Click on the EXIT button to exit from the editor.

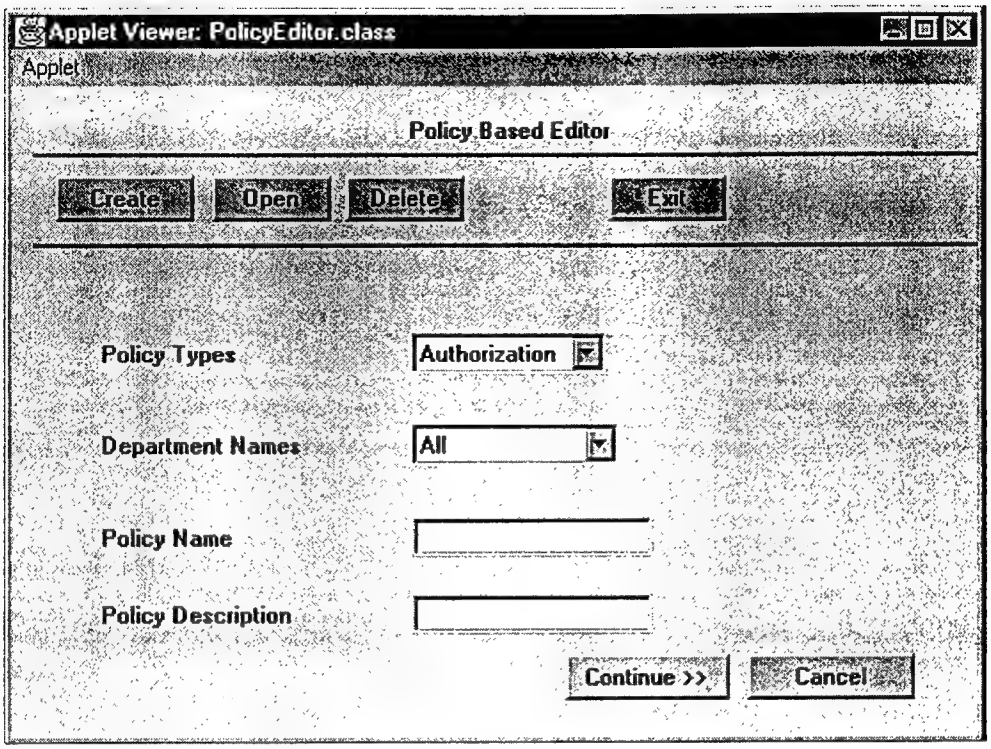


Figure 26. Creating a new policy

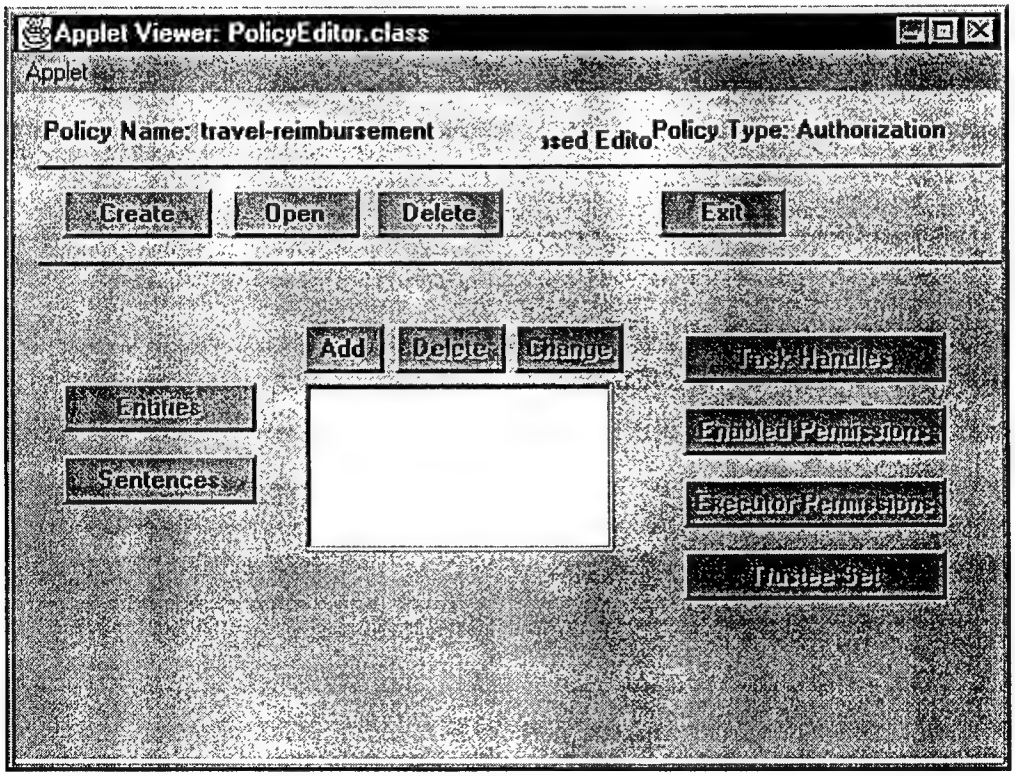


Figure 27. Defining entities for a newly created policy

Creating a Policy

To create a policy, click the CREATE button. This will result in the screen shown in Figure 26. To create a policy one has to indicate the type of policy to be created (for TBAC, all policies created will be of the type "Authorization"). This has to be followed by the department the policy belongs to, the name of the policy, and a short textual description of the policy.

Opening and Editing a Policy

To access and edit an existing policy, press the **OPEN** button at the opening screen (main menu). This will result in the screen shown in Figure 28. Select the policy type (this will be Authorization for TBAC authorization policies) and a list of existing policies of the chosen type will be displayed. Selecting a policy from the list and pressing the **LOAD POLICY** button on the palette will result in the loading of the various entities and sentences for that particular policy as shown in Figure 29.

The name of the opened policy (the order processing policy defined in Section 3.1.2.5) is displayed on the left top corner of the screen. The user now has a choice of seeing the various entities defined for the policy or the various sentences by clicking the **ENTITIES** or **SENTENCES** buttons respectively. Selecting entities will result in the screen shown in Figure 30 that displays all defined authorization steps for the policy. The user can now add a new authorization-step to the policy, delete an authorization-step or change the definition of a selected authorization-step. Also, the user can view the components, namely the task-handle, enabled-permissions, executor-permissions, and trustee-set of an authorization-step.

Alternatively, the user can select the sentences of the policy and this will result in the screen shown in Figure 31 that displays the existing sentences defined in the policy. The user may now add a new sentence, edit (change) an existing sentence or delete an existing sentence. A user may also visualize the sentences in the policy by pressing the

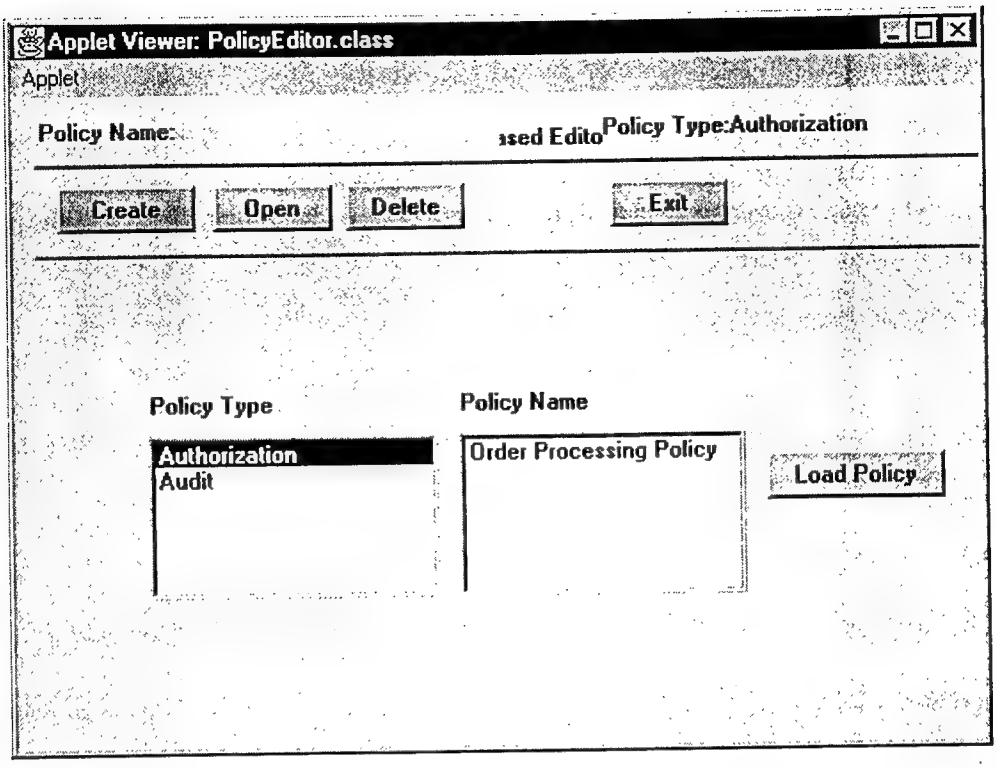


Figure 28. Selecting a policy to open

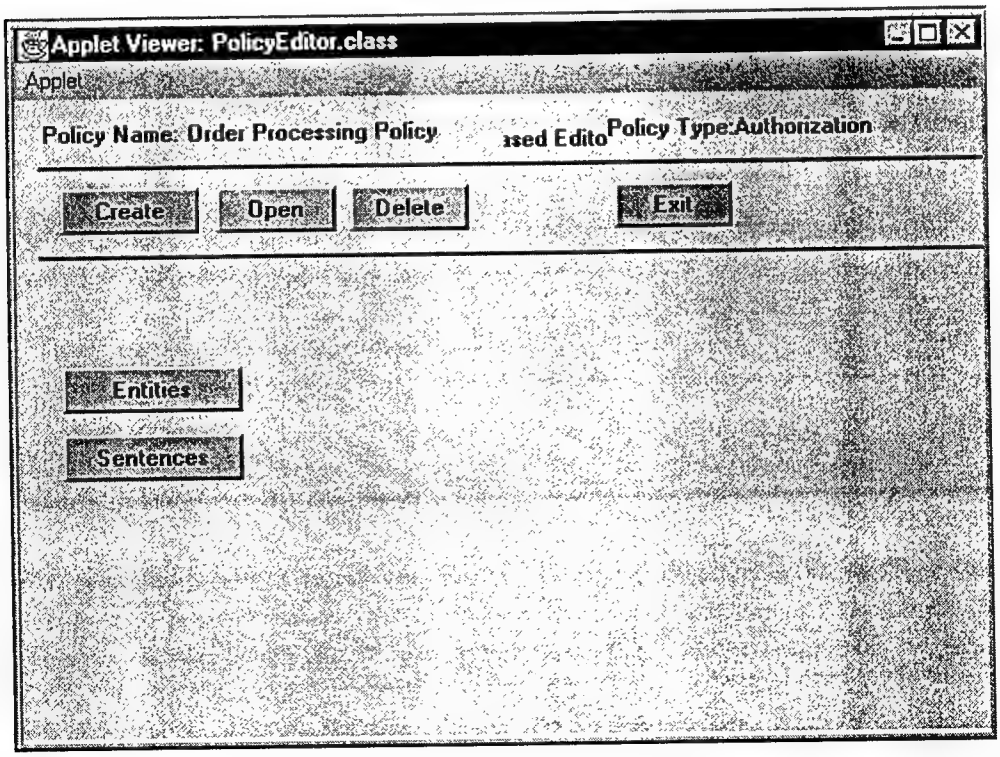


Figure 29. Browsing an opened policy

VISUALIZE button. As shown in screen Figure 32, the editor will then display the visual language representations of the various policy sentences.

There is also a help facility that gives the meanings of the various composite icons used in the visual sentences of policies. Click the **HELP** button on any screen displaying a visual sentence and the help facility is invoked (see Figure 33). This facility allows a user to scroll through the composite icons that are used to depict various states of authorization steps.

Deleting a Policy

To delete a policy, click the DELETE button at the main screen. And select the policy to be deleted (as shown in Figure 34).

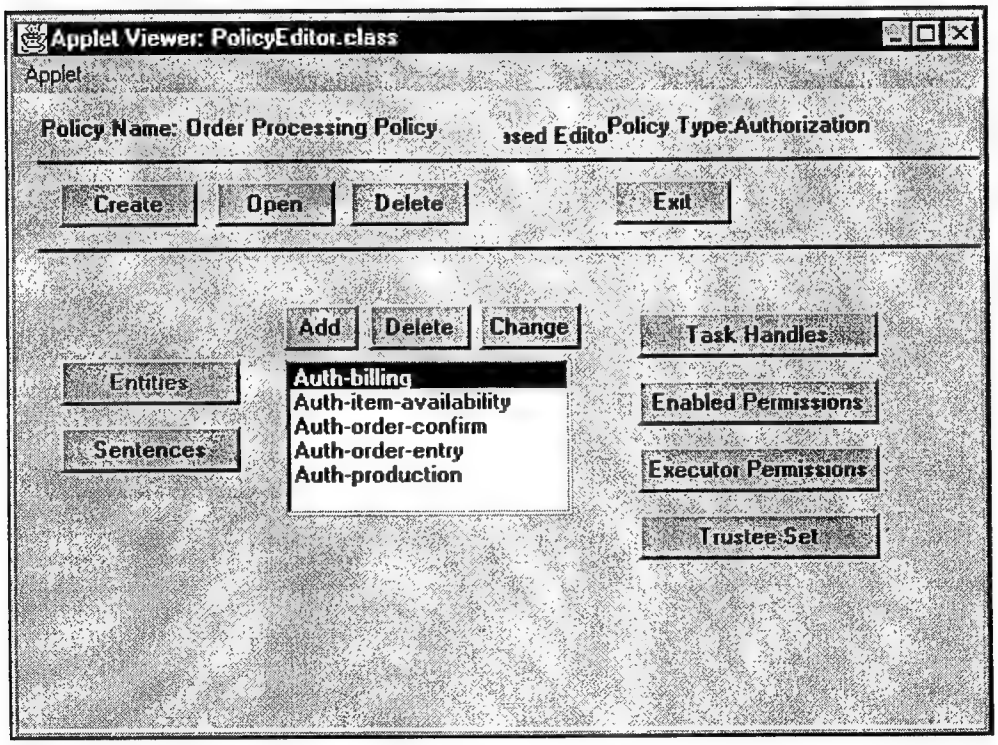


Figure 30. Viewing the entities of an open policy

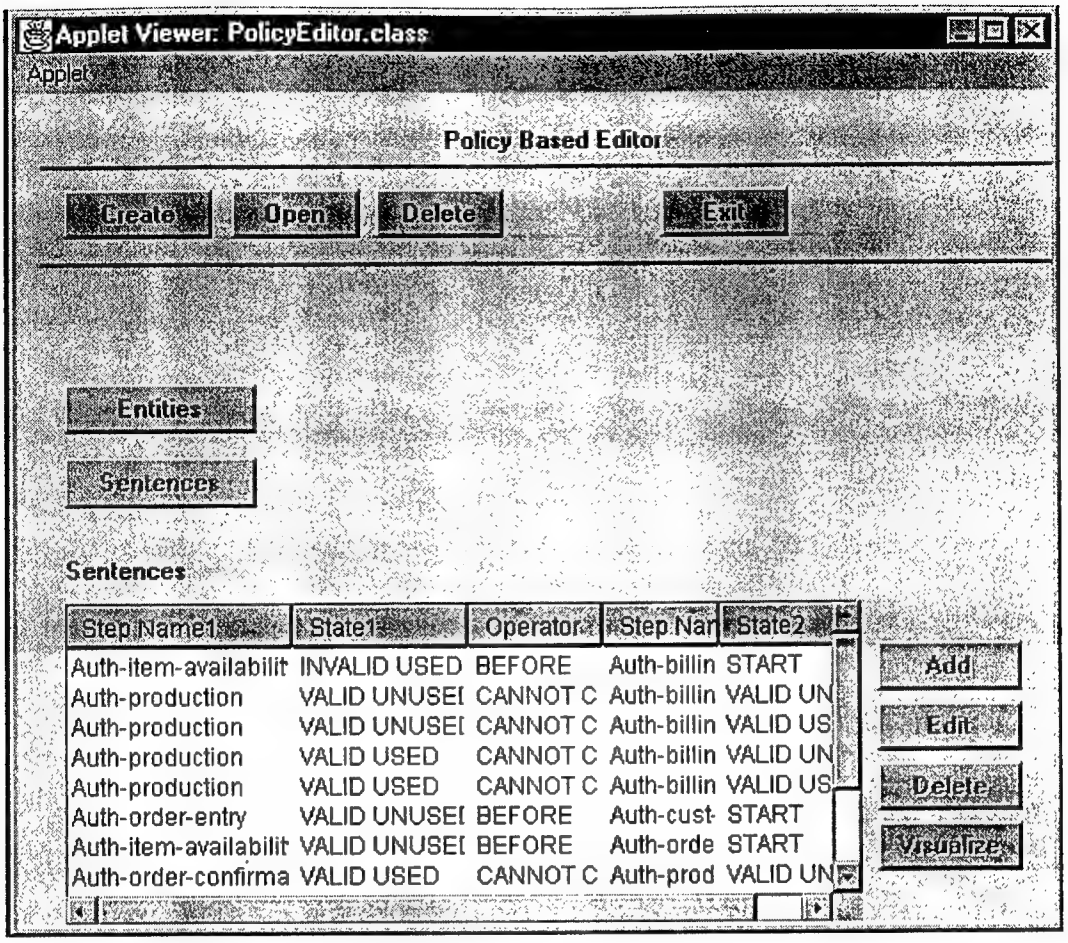


Figure 31. Viewing the sentences of an open policy

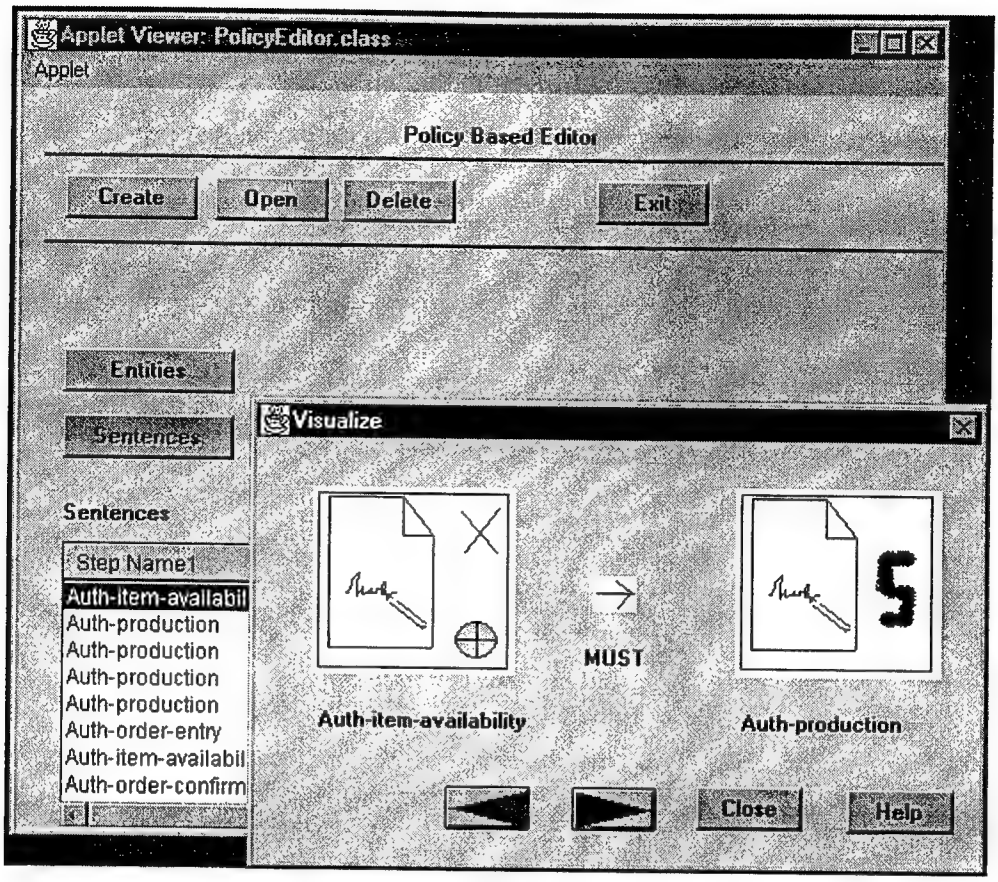


Figure 32. Visualizing policies

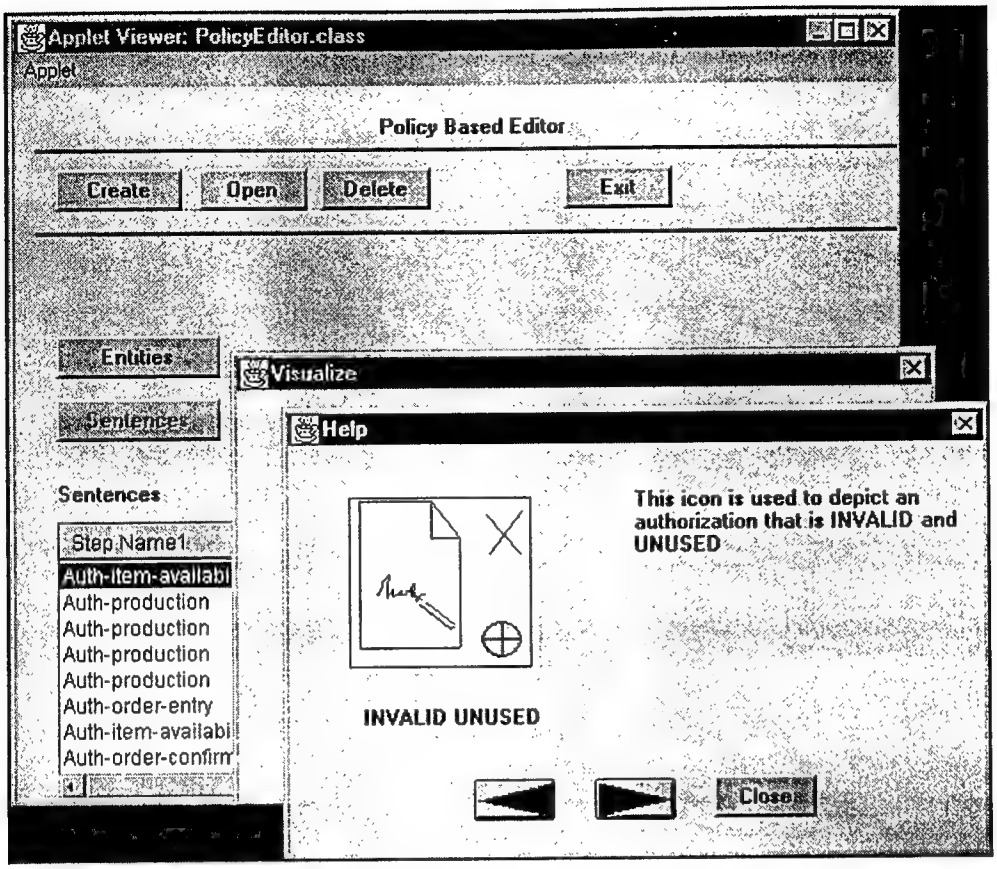


Figure 33. The help facility to visualize sentences

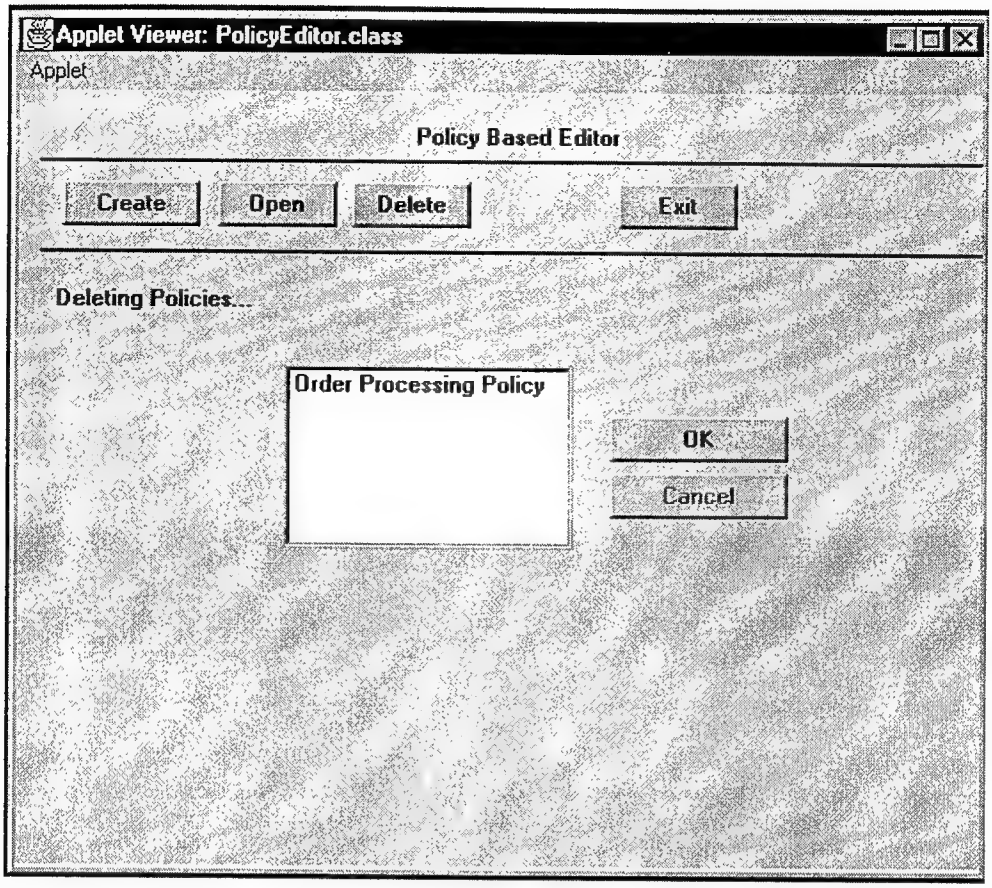


Figure 34. Deleting a policy

3.3.4.4 Configuring the authorization manager

Once the authorization server is installed, it needs to be configured. This is simple, and involves editing the **server.ini** file in the **bin** folder. The **server.ini** has two entries: The first is a TCP/IP port number where all traffic to the server will be directed to. The second entry in this file is the IP address of the machine where the server resides. Once these entries are supplied, they will be used by the authorization server and other client programs that communicate with the server.

3.3.4.5 Communicating with the authorization server

All communications with the authorization server is accomplished through messages. For the purposes of testing and demonstrating the server, we built two simple graphical user interfaces (GUIs) that can be used to send messages to the server and receive responses back. These interfaces are shown in Figure 35 and Figure 36. The executor interface can be used by an executor trustee to invoke an authorization-step and to subsequently grant or deny the authorization-step. Once an authorization has been successfully invoked, the server returns an authorization confirmation number. The executor is also free to distribute this confirmation number to other requestor trustees who want to make use of the authorization. To make use of an authorization, a requestor presents his/her name (also called the trustee name), the authorization step name, the

permission required, and the confirmation number received from an executor. Based on these parameters, the authorization sever will determine if the permission can be granted, and if so will dynamically issue a call to a backend system (such as NT server) to change permissions on the object associated with the permission.

These executor and requestor interfaces are only meant to illustrate the basic capabilities of the server. Developers and systems programmers who incorporate TBAC into their environments may build more complex interfaces to communicate with the server or alternatively formulate appropriate APIs to assemble and send the relevant messages to the server. One can also communicate with the server by simply assembling messages and making the appropriate network calls to send and receive messages over network transport (such as using WINSOCK).

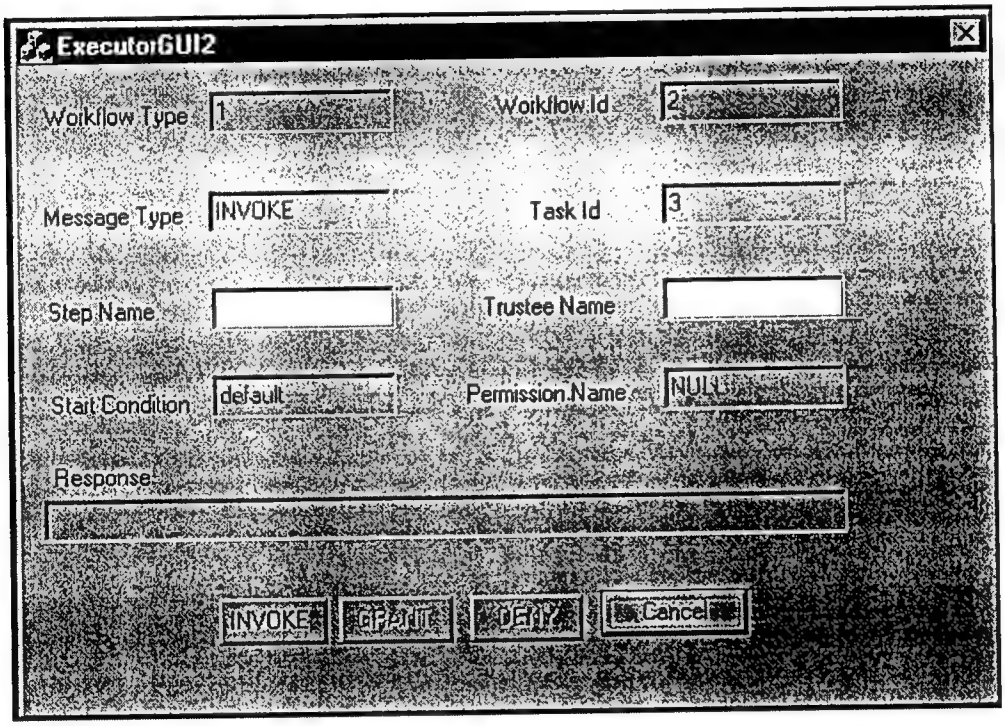


Figure 35. Executor interface to invoke, grant, and deny an authorization step

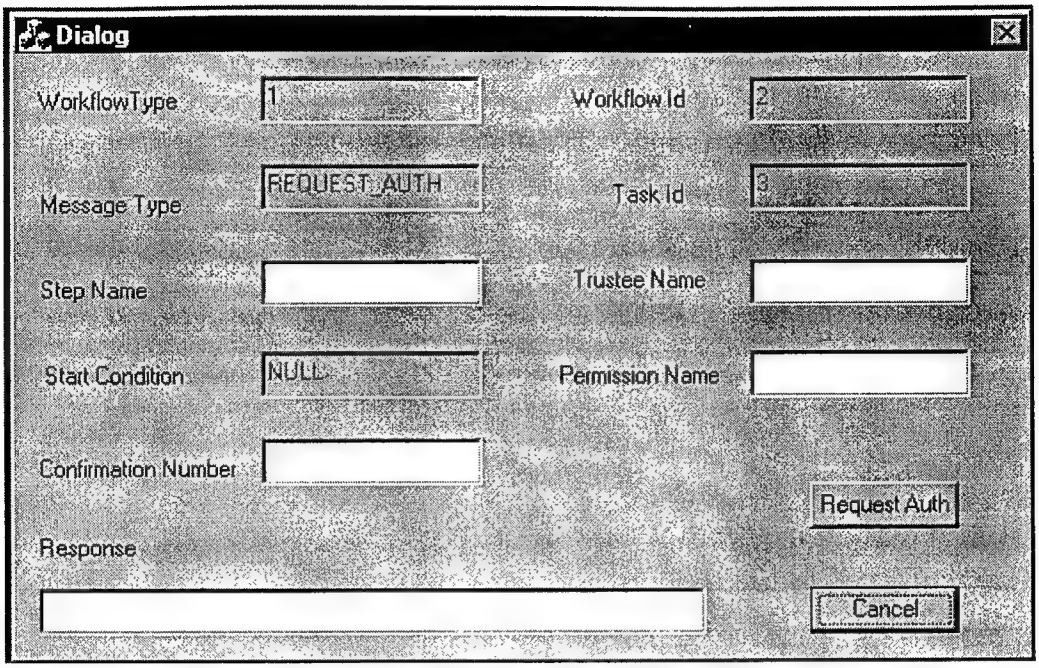


Figure 36. Requestor interface to to use an authorization

Message formats

The authorization manager understands the following message types

1. INVOKE – 0
2. REQUEST_AUTH - 1
3. GRANT – 2
4. DENY – 3

The general format of a message is the following:

ClientProcessId | WorkflowType | WflowId | TaskId | MsgType | StepName | TrusteeName | StartCondition | EnabledPermissionName | ConfirmationNumber | IP_Address_Of_Client |

Not all these fields contain useful data at all times. Only some particular fields are used during a particular type of message.

Messages of type INVOKE

Format:

WorkflowType | WflowId | TaskId | MsgType | StepName | TrusteeName | StartCondition

The client (either the Executor GUI or the Requester GUI) appends the following:

- Its own process id
- The "NULL" string value for the Enabled Permission Name
- The 0 value for the number
- The IP Address of the client itself

Messages of type GRANT or DENY

Format:

WorkflowType | WflowId | TaskId | MsgType | StepName | TrusteeName

For messages of type GRANT The MsgType should be 2 and for GRANT it should be 3

The client (either the Executor GUI or the Requester GUI) appends the following:

- Its own process id
- The "NULL" string value for the Enabled Permission Name
- The "NULL string value for the Start Condition
- The 0 value for the number
- The IP Address of the client itself

Messages of type REQUEST_AUTH

Format

WorkflowType | WflowId | TaskId | MsgType | StepName | TrusteeName | StartCondition | EnabledPermissionName

The client (either the Executor GUI or the Requester GUI) appends the following:

- Its own process id
- The IP Address of the client itself

3.3.4.6 Integration with the workflow system

The authorization server is built in such a way that it can be integrated easily with workflow management systems. The key functionality the authorization server expects from a workflow management system is the ability to send and receive messages. A secondary aspect of integration is the mechanism by which the interfaces for executor and requestor trustees are invoked from running workflow instances. These will vary from one workflow system to another.

For our prototype demonstration system, we have integrated the authorization server with Novell's Groupwise Workflow Professional workflow facility running on top of Novell's Groupwise messaging infrastructure. Figure 37 shows a simple workflow being defined with the Groupwise workflow authoring tool. The executor and requestor interfaces are incorporated into the workflow definition be attaching them as agents to tasks. This is shown in

Figure 38. The agent definition involves giving a name for the agent and specifying where the executable code (.exe file) for the agent resides. When the task is eventually routed to the appropriate recipient, it shows up as a work item in the recipient's inbox and task list. When the work item is selected from the task list, the user now has the option of "agents" tab and running any agents that were attached to the work item. This is shown in Figure 39. Thus, the executor of an authorization-step will have the appropriate executor agent attached to the work item. Running the agent will result in the invocation of the executor interface, which can be used to invoke, grant, and deny authorizations.

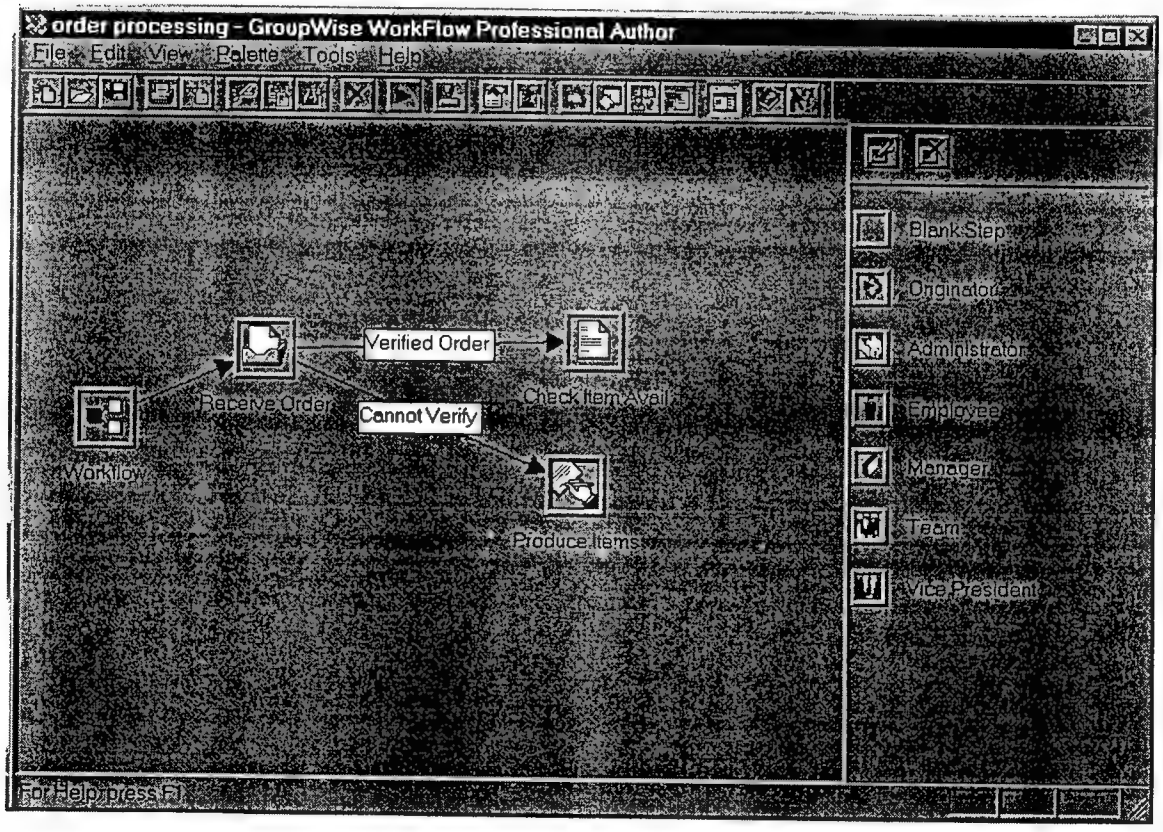


Figure 37. Creating a simple workflow with a workflow authoring tool

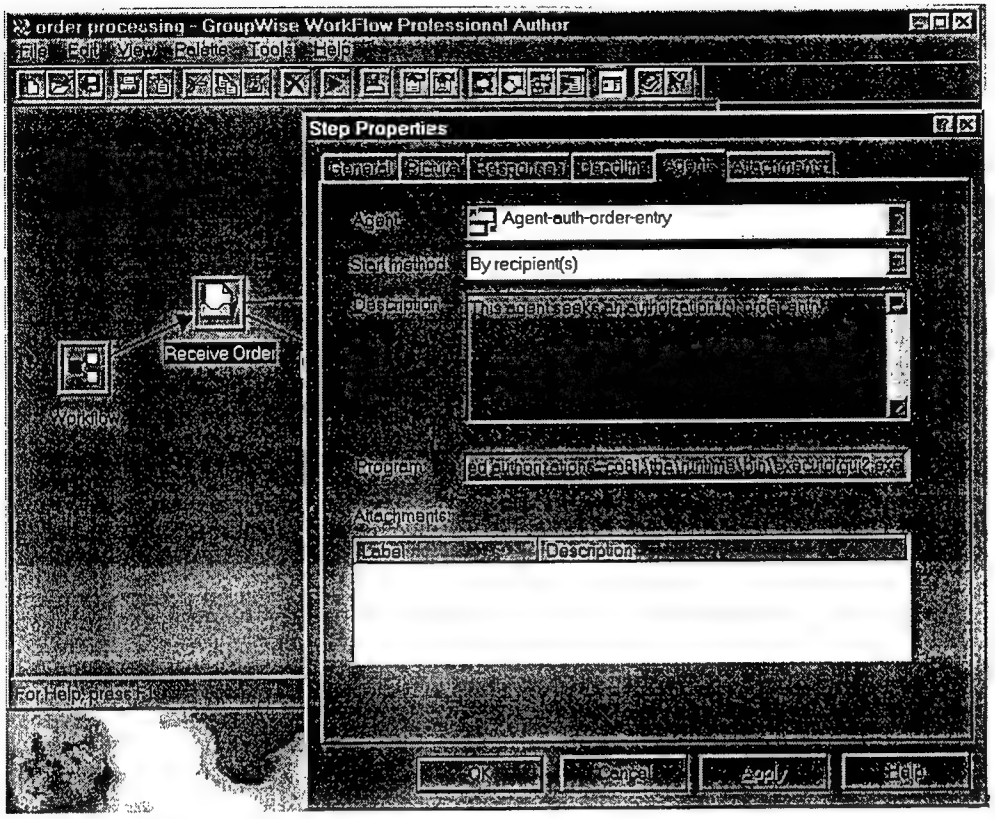


Figure 38. Defining an agent for a task in a workflow

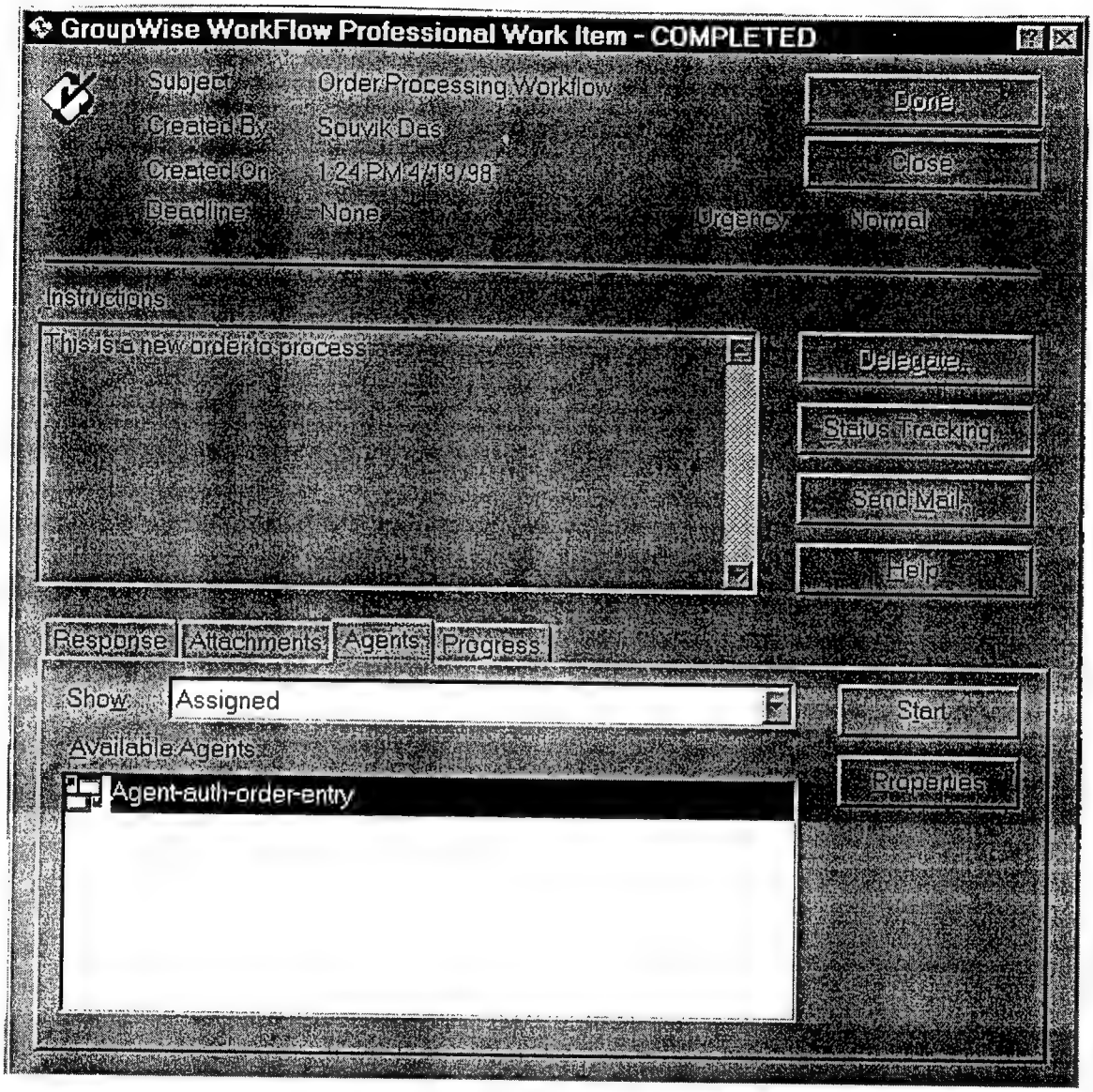


Figure 39. Invoking agents at runtime.

4. Conclusions

This final report documents our work on the TBAC project for the past two years of the contract. To summarize, any model of TBAC must support the notions of tasks, authorization-steps, and dependencies. With these concepts, one can model authorization policies that are common in many routine tasks, workflows, and applications. TBAC differs from traditional security models in that it is an active model, and as such, manages authorizations and permissions as tasks progress in accordance with some application logic. In TBAC models, permissions are dynamically activated and deactivated in response to events that occur in tasks, thereby enabling better task-based, need-to-know controls and security enforcement.

In future work, we hope to pursue the development of the entire family of TBAC models. We have encountered many challenges and learned many lessons during the course of the project. The greatest challenges arose from the integration of TBAC with the Windows NT security management APIs. We also learned that setting up and integrating a workflow system into an enterprise computing backbone is an arduous process with many difficulties arising from security and network settings. We hope to apply the lessons learned over the course of the project in building a more robust implementation in the future. We also hope to prototype various workflow scenarios in military and DOD applications, applying TBAC ideas to manage the various authorizations. During the TBAC project, we considered the JFACC program as a testbed for such prototyping. However, while JFACC has been investigating the use of workflows to manage the air campaign process, no sufficient details of workflow requirements or models were available from this domain in a timely enough manner to influence our work.

4. References

1. R. Strens and J. Dobson, How Responsibility Modeling leads to Security Requirements. Proceedings of the Second New Security Paradigms Workshop, Little Compton, Rhode Island, IEEE Press, 1993.
2. L.J. LaPadula and J.G Williams. Towards a Universal Integrity Model. Proceedings of the IEEE Computer Security Foundations Workshop, New Hampshire, IEEE Press, 1991.
3. R.K. Thomas and R.S. Sandhu. Towards a Task-based Paradigm for Flexible and Adaptable Access Control in Distributed Applications. Proceedings of the Second New Security Paradigms Workshop, Little Compton, Rhode Island, IEEE Press, 1993.
4. R.K. Thomas and R.S. Sandhu. Conceptual Foundations for A Model of Task-based Authorizations. Proceedings of the IEEE Computer Security Foundations Workshop, New Hampshire, IEEE Press, 1994.
5. R.K. Thomas and R.S. Sandhu. "Task-based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management," *Proceedings of the IFIP 11.3 Workshop on Database Security*, Lake Tahoe, California, August 1997.
6. S.K. Chang et. al. Visual-Language System for User Interfaces, IEEE Software, March, 1995.
7. S.K. Chang, G. Polese, R. Thomas, and S. Das. "A Visual Language for Authorization Modeling," Proceedings of IEEE Symposium on Visual Languages - VL97, Capri Island, Italy, September 23-27, 1997.
8. J. Klein. Advanced Rule Driven Transaction Management. Proceedings of the IEEE Compcon Conference, 1991.
9. D.E. Bell and L.J. LaPadula. Secure Computer Systems: Unified exposition and multics interpretation. EDS-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.
10. M.H. Harrison, W.L. Ruzzo and J.D. Ullman. Protection in Operating Systems. Communications of the ACM, 19(8), pages 461-471, 1976.
11. R.S. Sandhu. The Typed Access Control Model, Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 1992, pages 122-136.
12. V. Atluri and W. Huang. An Authorization Model for Workflows, Proceedings of the Fourth European Symposium on Research in Computer Security, Rome, Italy, September pages 25-27, 1996.
13. R.S. Sandhu. Transaction Control Expressions for Separation of Duties, Proceedings of the Fourth Computer Security Applications Conference, pages 282-286, 1988.

14. M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows, In Modern Database Systems: The Object Model, Interoperability, and beyond, W. Kim, Ed., Addison-Wesley / ACM Press, 1994.

15. D. Georgakopoulos, M. Hornick, and A. Sheth, An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure, Distributed and Parallel databases, Vol. 3, pages 119-153, 1995.

16. M. Abrams, K. Eggers, L. LaPadula, and I. Olson. A Generalized Framework for Access Control: An Informal Description, Proceedings of the 13th NIST-NCSC National Computer Security framework, 1990, pages 135-143.

17. M. Abrams, J. Heaney, O. King, L. LaPadula, M. Lazear and I. Olson. Generalized Framework for Access Control: Toward prototyping the Orgcon Policy, Proceedings of the 14th NIST-NCSC National Computer Security framework, 1991, pages 257-266.

18. W. Boebert and R. Kain. A Practical Alternative to Hierarchical Integrity Policies, Proceedings of the NBS-NCSC National Computer security Conference, 1985, pages 18-27.

*U.S. GOVERNMENT PRINTING OFFICE: 1999-610-130-81120

JOSEPH V. GIORDANO 3
AFRL/IFGB
525 BROOKS RD
ROME, NY 13441-4505

ODYSSEY RESEARCH ASSOCIATES 5
33 THORNWOOD DRIVE, SUITE 500
ITHACA, NY 14850

AFRL/IFOIL 1
TECHNICAL LIBRARY
26 ELECTRONIC PKY
ROME NY 13441-4514

ATTENTION: DTIC-OCC 2
DEFENSE TECHNICAL INFO CENTER
8725 JOHN J. KINGMAN ROAD, STE 0944
FT. BELVOIR, VA 22060-6218

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
3701 NORTH FAIRFAX DRIVE 2
ARLINGTON VA 22203-1714

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.